ModelSim®

Xilinx Edition II

User's Manual

Version 5.7c

Published: 11/Mar/03

The world's most popular HDL simulator

ModelSim is produced by Model Technology™, a Mentor Graphics Corporation company. Copying, duplication, or other reproduction is prohibited without the written consent of Model Technology.

The information in this manual is subject to change without notice and does not represent a commitment on the part of Model Technology. The program described in this manual is furnished under a license agreement and may not be used or copied except in accordance with the terms of the agreement. The online documentation provided with this product may be printed by the end-user. The number of copies that may be printed is limited to the number of licenses purchased.

ModelSim is a registered trademark and Signal Spy, TraceX, ChaseX and Model Technology are trademarks of Mentor Graphics Corporation. PostScript is a registered trademark of Adobe Systems Incorporated. UNIX is a registered trademark of AT&T in the USA and other countries. FLEXlm is a trademark of Globetrotter Software, Inc. IBM, AT, and PC are registered trademarks, AIX and RISC System/6000 are trademarks of International Business Machines Corporation. Windows, Microsoft, and MS-DOS are registered trademarks of Microsoft Corporation. OSF/Motif is a trademark of the Open Software Foundation, Inc. in the USA and other countries. SPARC is a registered trademark and SPARCstation is a trademark of SPARC International, Inc. Sun Microsystems is a registered trademark, and Sun, SunOS and OpenWindows are trademarks of Sun Microsystems, Inc. All other trademarks and registered trademarks are the properties of their respective holders.

Copyright © 1990 -2003, Model Technology, a Mentor Graphics Corporation company. All rights reserved. Confidential. Online documentation may be printed by licensed customers of Model Technology and Mentor Graphics for internal business purposes only.

ModelSim support

Support for ModelSim is available from your FPGA vendor. See the About ModelSim dialog box (accessed via the Help menu) for contact information.

Table of Contents

1 - Introduction (UM-13)

Standards supported . . . UM-14
Assumptions . . . UM-14
Sections in this document . . . UM-14
What is an "HDL item" . . . UM-16
Text conventions . . . UM-16

2 - Projects (UM-17)

Introduction . . . UM-18
- How do projects differ from pre-5.5 versions? . . . UM-19
- Project conversion between versions . . . UM-19

Getting started with projects . . . UM-20
- Step 1 — Creating a new project . . . UM-20
- Step 2 — Adding items to the project . . . UM-21
- Step 3 — Compiling the files . . . UM-24
- Step 4 — Simulating a design . . . UM-25
- Other basic project operations . . . UM-25

The Project tab . . . UM-26
- Project tab context menu . . . UM-27

Changing compile order . . . UM-28
- Grouping files . . . UM-29

Creating a Simulation Configuration . . . UM-30
Organizing projects with folders . . . UM-32
Setting compiler options . . . UM-34
Accessing projects from the command line . . . UM-35

3 - Design libraries (UM-37)

Design library contents . . . UM-38
Design library types . . . UM-39
Working with design libraries . . . UM-40
- Managing library contents . . . UM-41
- Assigning a logical name to a design library . . . UM-43
- Moving a library . . . UM-44

Specifying the resource libraries . . . UM-45
- Predefined libraries . . . UM-46
- Alternate IEEE libraries supplied . . . UM-46
- Regenerating your design libraries . . . UM-47

Importing FPGA libraries . . . UM-48

4 - VHDL simulation (UM-49)

Compiling VHDL designs . . . UM-50
- Invoking the VHDL compiler . . . UM-50
- Dependency checking . . . UM-50
- Range and index checking . . . UM-50

Simulating VHDL designs . . . UM-52
- Simulator resolution limit . . . UM-52
- Delta delays . . . UM-53

Using the TextIO package . . . UM-55
- Syntax for file declaration . . . UM-55
- Using STD_INPUT and STD_OUTPUT within ModelSim . . . UM-56

TextIO implementation issues . . . UM-57
- Reading and writing hexadecimal numbers . . . UM-58
- Dangling pointers . . . UM-58
- The ENDLINE function . . . UM-58
- The ENDFILE function . . . UM-58
- Using alternative input/output files . . . UM-59
- Providing stimulus . . . UM-59

VITAL specification and source code . . . UM-60

VITAL packages . . . UM-60

ModelSim VITAL compliance . . . UM-60
- VITAL compliance checking . . . UM-60

Compiling and simulating with accelerated VITAL packages . . . UM-61

Util package . . . UM-62
- get_resolution . . . UM-62
- init_signal_driver() . . . UM-63
- init_signal_spy() . . . UM-63
- signal_force() . . . UM-63
- signal_release() . . . UM-63
- to_real() . . . UM-64
- to_time() . . . UM-65

5 - Verilog simulation (UM-67)

Compilation . . . UM-69
- Incremental compilation . . . UM-70
- Library usage . . . UM-72
- Verilog-XL compatible compiler arguments . . . UM-73
- Verilog-XL ‘uselib compiler directive . . . UM-74

Simulation . . . UM-76
- Simulator resolution limit . . . UM-77
- Event ordering in Verilog designs . . . UM-79
- Negative timing check limits . . . UM-83
- Verilog-XL compatible simulator arguments . . . UM-86

Cell libraries . . . UM-87
- Delay modes . . . UM-87

System tasks UM-89
IEEE Std 1364 system tasks UM-89
Verilog-XL compatible system tasks UM-92
ModelSim Verilog system tasks UM-94
Compiler directives UM-95
IEEE Std 1364 compiler directives UM-95
Verilog-XL compatible compiler directives UM-96
Verilog PLI/VPI UM-97
Registering PLI applications UM-97
Registering VPI applications UM-99
Compiling and linking PLI/VPI C applications UM-101
Compiling and linking PLI/VPI C++ applications UM-102
Specifying the PLI/VPI file to load UM-103
PLI example UM-104
VPI example UM-105
The PLI callback reason argument UM-106
The sizetf callback function UM-107
PLI object handles UM-107
Third party PLI applications UM-108
Support for VHDL objects UM-109
IEEE Std 1364 ACC routines UM-110
IEEE Std 1364 TF routines UM-111
Verilog-XL compatible routines UM-113
64-bit support in the PLI UM-113
PLI/VPI tracing UM-113
Debugging PLI/VPI application code UM-115

6 - WLF files (datasets) and virtuals (UM-117)

WLF files (datasets) UM-118
Saving a simulation to a WLF file UM-119
Opening datasets UM-119
Viewing dataset structure UM-120
Managing multiple datasets UM-121
Saving at intervals with Dataset Snapshot UM-123
Virtual Objects (User-defined buses, and more) UM-125
Virtual signals UM-125
Virtual functions UM-126
Virtual regions UM-127
Virtual types UM-127
Dataset, WLF file, and virtual commands UM-128

7 - Graphic interface (UM-129)

Window overview UM-130
Common window features UM-131
Quick access toolbars UM-132
Drag and Drop UM-132

Command history UM-132
Automatic window updating UM-133
Finding names UM-133
Sorting HDL items UM-133
Saving window layout UM-134
Context menus UM-134
Menu tear off UM-134
Tree window hierarchical view UM-135

Main window UM-137
Workspace UM-138
Transcript UM-139
The Main window menu bar UM-140
The Main window toolbar UM-145
The Main window status bar UM-147
Mouse and keyboard shortcuts UM-147

Dataflow window UM-149
Adding items to the window UM-149
Links to other windows UM-150
Dataflow window menu bar UM-150
The Dataflow window toolbar UM-153
Exploring the connectivity of your design UM-156
Zooming and panning UM-158
Tracing events (causality) UM-159
Tracing the source of an unknown (X) UM-160
Finding items by name in the Dataflow window UM-161
Saving the display UM-162
Configuring page setup UM-164
Symbol mapping UM-165
Configuring window options UM-166

List window UM-168
HDL items you can view UM-168
Adding HDL items to the List window UM-169
The List window menu bar UM-170
Editing and formatting HDL items in the List window UM-172
Combining items in the List window UM-174
Setting List window display properties UM-175
Finding items by name in the List window UM-177
Setting time markers in the List window UM-178
Saving List window data to a file UM-179
List window keyboard shortcuts UM-180

Process window UM-181
The Process window menu bar UM-182

Signals window UM-183
The Signals window menu bar UM-184
Filtering the signal list UM-185
Forcing signal and net values UM-186
Adding HDL items to the Wave and List windows or a WLF file UM-187
Finding HDL items in the Signals window UM-188
Setting signal breakpoints UM-189

Defining clock signals UM-189
Source window UM-191
The Source window menu bar UM-192
The Source window toolbar UM-194
Setting file-line breakpoints UM-197
Checking HDL item values and descriptions UM-197
Finding and replacing in the Source window UM-197
Setting tab stops in the Source window UM-198
Structure window UM-199
Structure window menu bar UM-200
Structure window context menu UM-201
Finding items in the Structure window UM-202
Variables window UM-203
The Variables window menu bar UM-204
Finding HDL items in the Variables window UM-205
Wave window UM-206
Pathname pane UM-206
Values pane UM-207
Waveform pane UM-207
Cursor panes UM-207
HDL items you can view UM-207
Adding HDL items in the Wave window UM-208
The Wave window menu bar UM-209
The Wave window toolbar UM-212
Using dividers UM-215
Splitting Wave window panes UM-216
Combining items in the Wave window UM-217
Displaying drivers of the selected waveform UM-218
Editing and formatting HDL items in the Wave window UM-219
Setting Wave window display properties UM-222
Setting signal breakpoints UM-224
Finding items by name or value in the Wave window UM-225
Using time cursors in the Wave window UM-226
Examining waveform values UM-228
Zooming - changing the waveform display range UM-228
Saving zoom range and scroll position with bookmarks UM-229
Wave window mouse and keyboard shortcuts UM-231
Saving waveforms UM-233
Compiling with the graphic interface UM-238
Locating source errors during compilation UM-239
Setting default compile options UM-240
Simulating with the graphic interface UM-245
Design tab UM-245
VHDL tab UM-247
Verilog tab UM-249
Libraries tab UM-250
SDF tab UM-251
Options tab UM-253
Setting default simulation options UM-254

Creating and managing breakpoints UM-258
Signal breakpoints UM-258
File-line breakpoints UM-258
Breakpoints dialog UM-259
Miscellaneous tools and add-ons UM-262
The GUI Expression Builder UM-262
Language templates UM-264
Graphic interface commands UM-267

8 - Signal Spy (UM-269)

Introduction UM-270
init_signal_driver UM-271
init_signal_spy UM-274
signal_force UM-276
signal_release UM-278
$init_signal_driver UM-280
$init_signal_spy UM-283
$signal_force UM-285
$signal_release UM-287

9 - Standard Delay Format (SDF) Timing Annotation (UM-289)

Specifying SDF files for simulation UM-290
Instance specification UM-290
SDF specification with the GUI UM-291
Errors and warnings UM-291
VHDL VITAL SDF UM-292
SDF to VHDL generic matching UM-292
Resolving errors UM-293
Verilog SDF UM-294
The $sdf_annotate system task UM-294
SDF to Verilog construct matching UM-295
Optional edge specifications UM-298
Optional conditions UM-299
Rounded timing values UM-299
SDF for Mixed VHDL and Verilog Designs UM-300
Interconnect delays UM-300
Disabling timing checks UM-300
Troubleshooting UM-301
Mistaking a component or module name for an instance label UM-302
Forgetting to specify the instance UM-302

10 - Value Change Dump (VCD) Files (UM-303)

ModelSim VCD commands and VCD tasks UM-304
Creating a VCD file UM-306
Flow for four-state VCD file UM-306
Flow for extended VCD file UM-306
Case sensitivity UM-306
Resimulating a design from a VCD file UM-307
A VCD file from source to output UM-309
VCD simulator commands UM-309
VCD output UM-310
Capturing port driver data UM-312
Supported TSSI states UM-312
Strength values UM-313
Port identifier code UM-313
Example VCD output from vcd dumpports UM-314

11 - Tcl and macros (DO files) (UM-315)

Tcl features within ModelSim UM-316
Tcl References UM-316
Tcl commands UM-317
Tcl command syntax UM-318
if command syntax UM-320
set command syntax UM-321
Command substitution UM-321
Command separator UM-322
Multiple-line commands UM-322
Evaluation order UM-322
Tcl relational expression evaluation UM-322
Variable substitution UM-323
System commands UM-323
List processing UM-324
ModelSim Tcl commands UM-324
ModelSim Tcl time commands UM-325
Conversions UM-325
Relations UM-325
Arithmetic UM-326
Tcl examples UM-327
Example 2 . . . . . UM-328
Macros (DO files) UM-331
Using Parameters with DO files UM-331
Making macro parameters optional UM-332
Useful commands for handling breakpoints and errors UM-333

A - ModelSim variables (UM-335)

Variable settings report UM-336
Personal preferences UM-336
Returning to the original ModelSim defaults UM-337
Environment variables UM-337
- Creating environment variables in Windows UM-339
- Referencing environment variables within ModelSim UM-340
- Removing temp files (VSOUT) UM-340

Preference variables located in INI files UM-341
- [Library] library path variables UM-341
- [vcom] VHDL compiler control variables UM-342
- [vlog] Verilog compiler control variables UM-343
- [vsim] simulator control variables UM-344
- Commonly used INI variables UM-349

Preference variables located in Tcl files UM-352
- User-defined variables UM-352
- More preferences UM-352

Variable precedence UM-353
Simulator state variables UM-353
- Referencing simulator state variables UM-354
- Special considerations for the now variable UM-354

B - ModelSim shortcuts (UM-355)

Wave window mouse and keyboard shortcuts UM-356
List window keyboard shortcuts UM-357
Command shortcuts UM-358
Mouse and keyboard shortcuts in Main and Source windows UM-359
- Right mouse button UM-360

C - ModelSim messages (UM-361)

ModelSim message system UM-362
- Message format UM-362
- Getting more information UM-362

Suppressing warning messages UM-363
- Suppressing VCOM warning messages UM-363
- Suppressing VLOG warning messages UM-363
- Suppressing VSIM warning messages UM-363

Exit codes UM-364
Miscellaneous messages UM-366
- Empty port name warning UM-366
- Lock message UM-366
- Metavalue detected warning UM-366

Sensitivity list warning UM-367
Tcl Initialization error 2 . . . . . UM-367
Too few port connections UM-368
VSIM license lost UM-369

D - System initialization (UM-371)

Files accessed during startup UM-372
Environment variables accessed during startup UM-373
Initialization sequence UM-374

E - Tips and techniques (UM-377)

Running command-line and batch-mode simulations UM-378
Saving and viewing waveforms in batch mode UM-379
Setting up libraries for group use UM-379
Using a DO file to test for assertions UM-380
Locating assertion warnings UM-380
Sampling signals at a clock change UM-381
Configuring a List trigger with Expression Builder UM-382
Converting signal values to strings UM-384
Converting an integer into a bit_vector UM-385
Detecting infinite zero-delay loops UM-386
Referencing source files with location maps UM-387
Using location mapping UM-387
Pathname syntax UM-388
How location mapping works UM-388
Mapping with Tcl variables UM-388
Performance affected by scheduled events being cancelled UM-389
Modeling memory in VHDL UM-390

Index (UM-401)

1 - Introduction

Chapter contents

Standards supported UM-14
Assumptions UM-14
Sections in this document UM-14
What is an "HDL item" UM-16
Text conventions UM-16
What is an "HDL item" UM-16

This documentation was written for ModelSim version 5.7c for Microsoft Windows 98/Me/NT/2000/XP. If the ModelSim software you are using is a later release, check the README file that accompanied the software. Any supplemental information will be there.

Standards supported

ModelSim VHDL supports both the IEEE 1076-1987 and 1076-1993 VHDL, the 1164-1993 *Standard Multivalue Logic System for VHDL Interoperability*, and the 1076.2-1996 *Standard VHDL Mathematical Packages* standards. Any design developed with ModelSim will be compatible with any other VHDL system that is compliant with either IEEE Standard 1076-1987 or 1076-1993.

ModelSim Verilog is based on IEEE Std 1364-1995 and a partial implementation of 1364-2001 *Standard Hardware Description Language Based on the Verilog Hardware Description Language* (see */<install_dir>/modeltech/docs/technotes/vlog_2001.note* for implementation details). The Open Verilog International *Verilog LRM version 2.0* is also applicable to a large extent. Both PLI (Programming Language Interface) and VCD (Value Change Dump) are supported for ModelSim PE and SE users.

In addition, all products support SDF 1.0 through 3.0, VITAL 2.2b, VITAL'95 – IEEE 1076.4-1995, and VITAL 2000 – IEEE 1076.4-2000.

Assumptions

We assume that you are familiar with the use of your operating system. If you are not familiar with Microsoft Windows, we recommend that you work through the tutorials provided with MS Windows before using ModelSim.

We also assume that you have a working knowledge of VHDL and Verilog. Although ModelSim is an excellent tool to use while learning HDL concepts and practices, this document is not written to support that goal.

Finally, we make the assumption that you have worked the appropriate lessons in the *ModelSim Tutorial* and are therefore familiar with the basic functionality of ModelSim. The *ModelSim Tutorial* is available from the ModelSim **Help** menu.

Sections in this document

In addition to this introduction, you will find the following major sections in this document:

2 - Projects (UM-17)
This chapter discusses ModelSim "projects", a container for design files and their associated simulation properties.

3 - Design libraries (UM-37)
To simulate an HDL design using ModelSim, you need to know how to create, compile, maintain, and delete design libraries as described in this chapter.

4 - VHDL simulation (UM-49)
This chapter is an overview of compilation and simulation for VHDL within the ModelSim environment.

5 - Verilog simulation (UM-67)
This chapter is an overview of compilation and simulation for Verilog within the ModelSim environment.

6 - WLF files (datasets) and virtuals (UM-117)
This chapter describes datasets and virtuals - both methods for viewing and organizing simulation data in ModelSim.

7 - Graphic interface (UM-129)
This chapter describes the graphic interface available while operating ModelSim. ModelSim's graphic interface is designed to provide consistency throughout all operating system environments.

8 - Signal Spy (UM-269)
This chapter describes Signal Spy, a set of VHDL procedures and Verilog system tasks that let you monitor, drive, force, or release an item from anywhere in the hierarchy of a VHDL or mixed design.

9 - Standard Delay Format (SDF) Timing Annotation (UM-289)
This chapter discusses ModelSim's implementation of SDF (Standard Delay Format) timing annotation. Included are sections on VITAL SDF and Verilog SDF, plus troubleshooting.

10 - Value Change Dump (VCD) Files (UM-303)
This chapter explains Model Technology's Verilog VCD implementation for ModelSim. The VCD usage is extended to include VHDL designs.

11 - Tcl and macros (DO files) (UM-315)
This chapter provides an overview of Tcl (tool command language) as used with ModelSim.

A - ModelSim variables (UM-335)
This appendix describes environment, system, and preference variables used in ModelSim.

B - ModelSim shortcuts (UM-355)
This appendix describes ModelSim keyboard and mouse shortcuts.

C - ModelSim messages (UM-361)
This appendix describes ModelSim error and warning messages.

D - System initialization (UM-371)
This appendix describes what happens during ModelSim startup.

E - Tips and techniques (UM-377)
This appendix contains a collection of ModelSim usage examples taken from our manuals and tech support solutions.

What is an "HDL item"

Because ModelSim works with both VHDL and Verilog, “HDL” refers to either VHDL or Verilog when a specific language reference is not needed. Depending on the context, “HDL item” can refer to any of the following:

VHDL	block statement, component instantiation, constant, generate statement, generic, package, signal, or variable
Verilog	function, module instantiation, named fork, named begin, net, task, or register variable

Text conventions

Text conventions used in this manual include:

italic text	provides emphasis and sets off filenames, path names, and design unit names
bold text	indicates commands, command options, menu choices, package and library logical names, as well as variables, dialog box selections, and language keywords
`monospace type`	monospace type is used for program and command examples
The right angle (>)	is used to connect menu choices when traversing menus as in: **File > Quit**
UPPER CASE	denotes file types used by ModelSim (e.g., DO, WLF, INI, MPF, PDF, etc.)

2 - Projects

Chapter contents

Introduction UM-18
What are projects? UM-18
What are the benefits of projects? UM-18
How do projects differ from pre-5.5 versions? UM-19
Project conversion between versions UM-19

Getting started with projects UM-20
Step 1 — Creating a new project UM-20
Step 2 — Adding items to the project UM-21
Step 3 — Compiling the files UM-21
Step 4 — Simulating a design UM-21
Other basic project operations UM-25

The Project tab UM-26
Sorting the list UM-26
Project tab context menu UM-27

Changing compile order UM-28
Auto-generating compile order UM-28
Grouping files UM-29

Creating a Simulation Configuration UM-30

Organizing projects with folders UM-32

Setting compiler options UM-34

Accessing projects from the command line UM-35

This chapter discusses ModelSim projects. Projects simplify the process of compiling and simulating a design and are a great tool for getting started with ModelSim.

Introduction

What are projects?

Projects are collection entities for HDL designs under specification or test. At a minimum, projects have a root directory, a work library, and "metadata" which are stored in a *.mpf* file located in a project's root directory. The metadata include compiler switch settings, compile order, and file mappings. Projects may also include:

- HDL source files or references to source files
- other files such as READMEs or other project documentation
- local libraries
- references to global libraries
- Simulation Configurations (see "Creating a Simulation Configuration" (UM-30)
- Folders (see "Organizing projects with folders" (UM-32))

Important: Project metadata are updated and stored *only* for actions taken within the project itself. For example, if you have a file in a project, and you compile that file from the command line rather than using the project menu commands, the project will not update to reflect any new compile settings.

What are the benefits of projects?

Projects offer benefits to both new and advanced users. Projects

- simplify interaction with ModelSim; you don't need to understand the intricacies of compiler switches and library mappings
- eliminate the need to remember a conceptual model of the design; the compile order is maintained for you in the project
- remove the necessity to re-establish compiler switches and settings at each session; these are stored in the project metadata as are mappings to HDL source files
- allow users to share libraries without copying files to a local directory; you can establish references to source files that are stored remotely or locally
- allow you to change individual parameters across multiple files; in previous versions you could only set parameters one file at a time
- enable "what-if" analysis; you can copy a project, manipulate the settings, and rerun it to observe the new results
- reload .ini variable settings every time the project is opened; in previous versions you had to quit ModelSim and restart the program to read in a new *.ini* file

How do projects differ from pre-5.5 versions?

Projects have improved a great deal from versions prior to 5.5. Some of the key differences include:

- A new interface eliminates the need to write custom scripts.
- You don't have to copy files into a specific directory; you can establish references to files in any location.
- You don't have to specify compiler switches; the automatic defaults will work for many designs. However, if you do want to customize the settings, you do it through a dialog box rather than by writing a script.
- All metadata (compiler settings, compile order, file mappings, etc.) are stored in the project *.mpf* file.

Note: Due to the significant changes, projects created in versions prior to 5.5 cannot be converted automatically. If you created a project in an earlier version, you will need to recreate it in versions later than 5.5. With the new interface even the most complex project should take less than 15 minutes to recreate. Follow the instructions in the ensuing pages to recreate your project.

Project conversion between versions

Projects are generally not backwards compatible for either number or letter releases. When you open a project created in an earlier version (e.g, you're using 5.6 and you open a project created in 5.5), you'll see a message warning that the project will be converted to the newer version. You have the option of continuing with the conversion or cancelling the operation.

As stated in the warning message, a backup of the original project is created before the conversion occurs. The backup file is named <*project name*>*.mpf.bak* and is created in the same directory in which the original project is located.

Getting started with projects

This section describes the four basic steps to working with a project.

Step 1 — Creating a new project (UM-20)

This creates a .mpf file and a working library.

Step 2 — Adding items to the project (UM-21)

Projects can reference or include HDL source files, folders for organization, simulations, and any other files you want to associate with the project. You can copy files into the project directory or simply create mappings to files in other locations.

Step 3 — Compiling the files (UM-24)

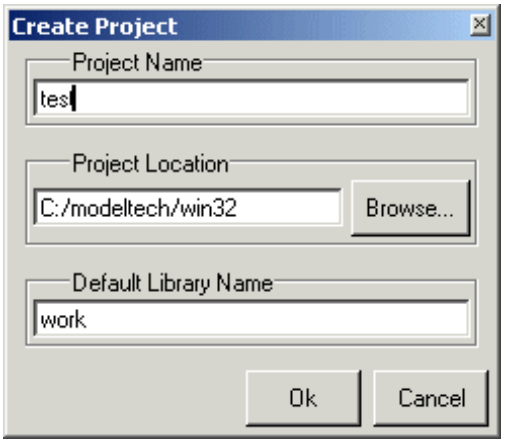
This checks syntax and semantics and creates the pseudo machine code ModelSim uses for simulation.

Step 4 — Simulating a design (UM-25)

This specifies the design unit you want to simulate and opens a structure tab in the Main window workspace.

Step 1 — Creating a new project

Select **File > New > Project** (Main window) to create a new project. This opens the **Create Project** dialog.

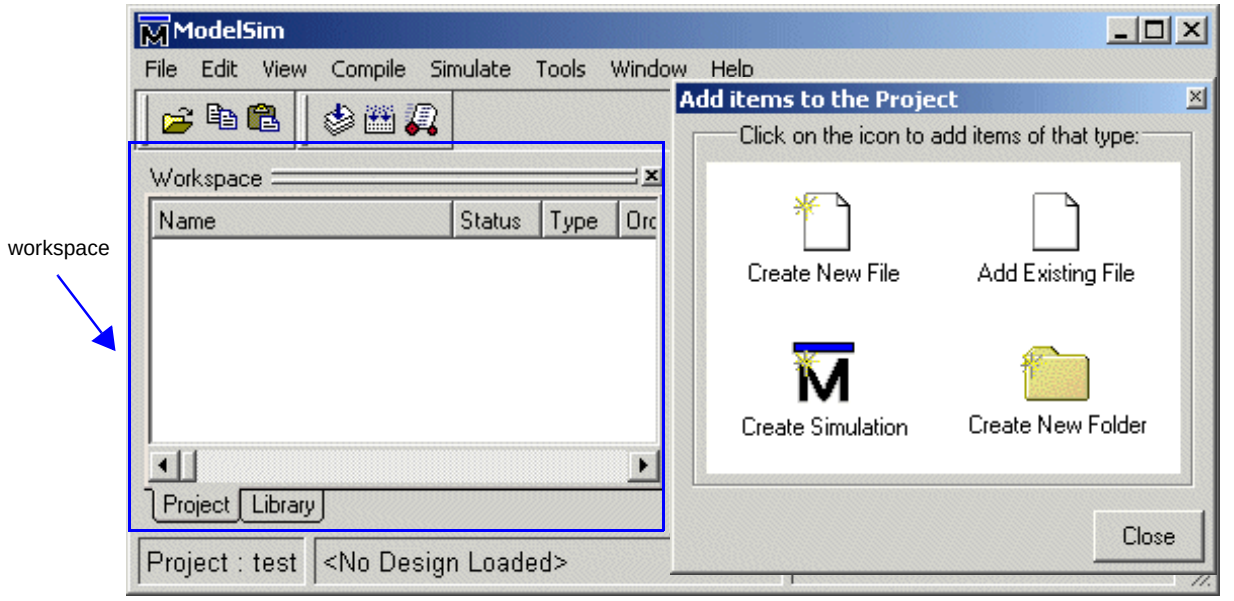


The dialog includes these options:

- **Project Name**
  The name of the new project.
- **Project Location**
  The directory in which the *.mpf* file will be created.
- **Default Library Name**
  The name of the working library. See "Design library types" (UM-39) for more details on work libraries. You can generally leave the **Default Library Name** set to "work." The

name you specify will be used to create a working library subdirectory within the Project Location.

After selecting OK, you will see a blank Project tab in the workspace area of the Main window and the **Add Items to the Project** dialog.



The name of the current project is shown at the bottom left corner of the Main window.

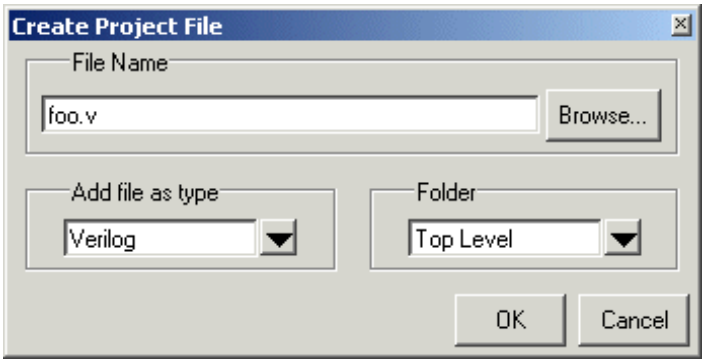
Step 2 — Adding items to the project

The **Add Items to the Project** dialog includes these options:

- **Create New File**
  Create a new VHDL, Verilog, Tcl, or text file using the Source window. See below for details.
- **Add Existing File**
  Add an existing file. See below for details.
- **Create Simulation**
  Create a Simulation Configuration that specifies source files and simulator options. See "Creating a Simulation Configuration" (UM-30) for details.
- **Create New Folder**
  Create an organization folder. See "Organizing projects with folders" (UM-32) for details.

Create New File

The **Create New File** command lets you create a new VHDL, Verilog, Tcl, or text file using the Source window. You can also access this command by selecting **File > Add to Project > New File** (Main window) or right-clicking



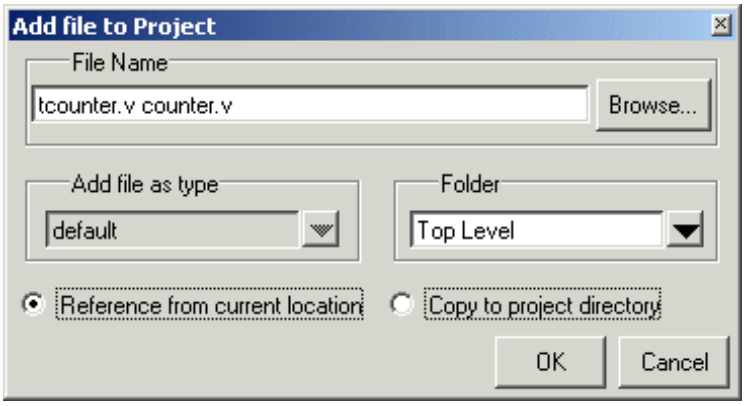
The **Create Project File** dialog includes these options:

- **File Name**
  The name of the new file.
- **Add file as type**
  The type of the new file. Select VHDL, Verilog, TCL, or text.
- **Folder**
  The organization folder in which you want the new file placed. You must first create folders in order to access them here. See "Organizing projects with folders" (UM-32) for details.

When you select OK, the Source window opens with an empty file, and the file is listed in the Project tab of the Main window workspace.

Add Existing File

You can also access this command by selecting **File** > **Add to Project** > **Existing File** (Main window) or by right-clicking



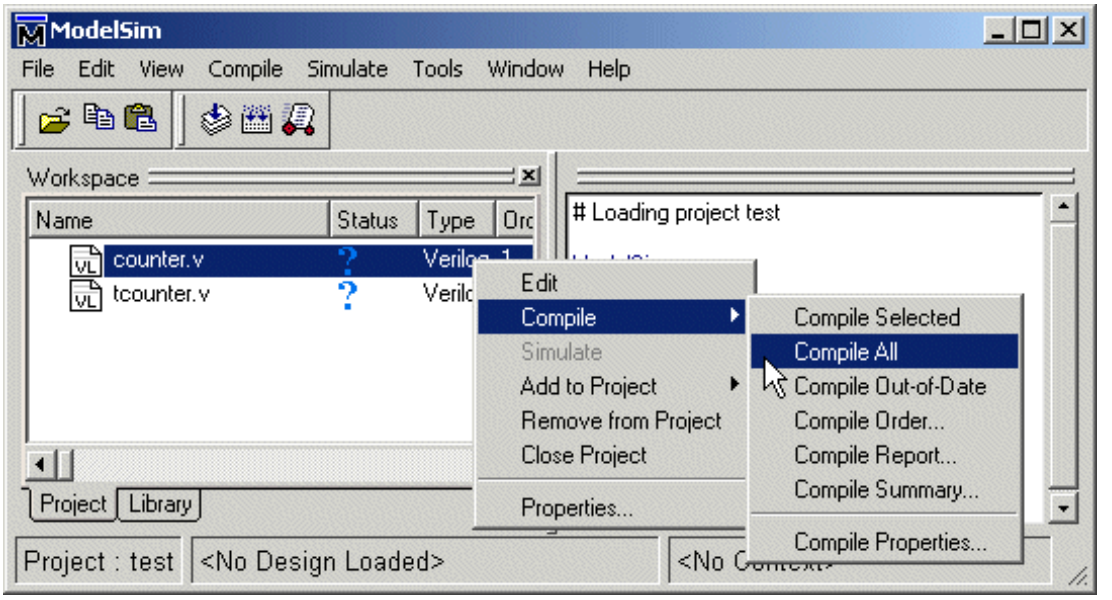
The **Add file to Project** dialog includes these options:

- **File Name**
  The name of the file to add. You can add multiple files at one time.
- **Add file as type**
  The type of the file. "Default" assigns type based on the file extension (e.g., *.v* is type Verilog).
- **Folder**
  The organization folder in which you want the file placed. You must first create folders in order to access them here. See "Organizing projects with folders" (UM-32) for details.
- **Reference from current location/Copy to project directory**
  Choose whether to reference the file from its current location or to copy it into the project directory.

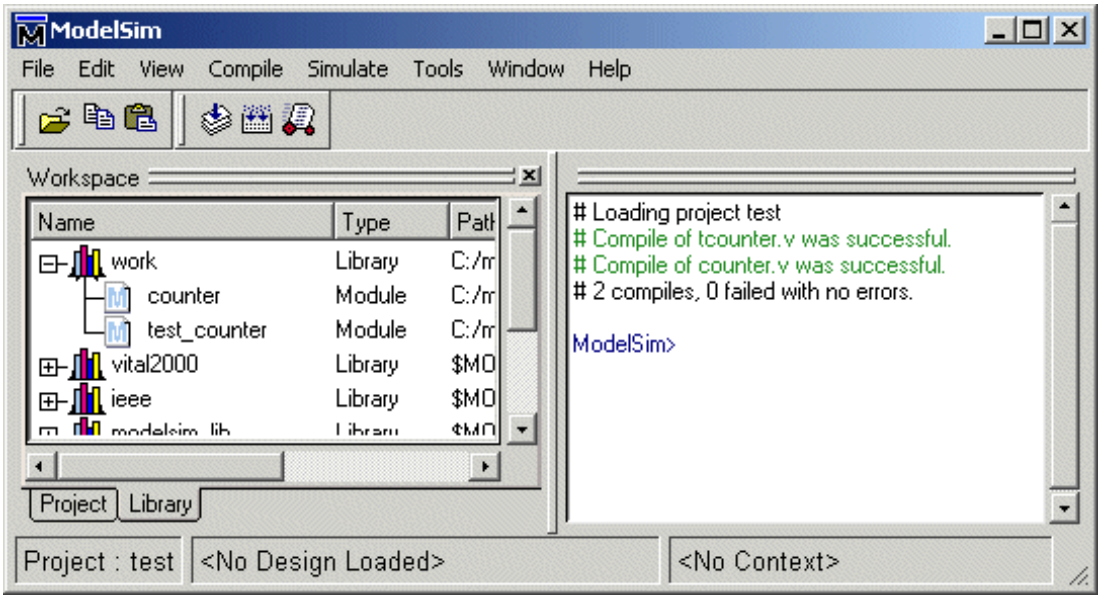
When you select OK, the file(s) is listed in the Project tab of the Main window workspace.

Step 3 — Compiling the files

The question marks next to the files in the Project tab denote either the files haven't been compiled into the project or the source has changed since the last compile. To compile the files, select **Compile** > **Compile All** (Main window) or right click in the Project tab and select **Compile** > **Compile All**.

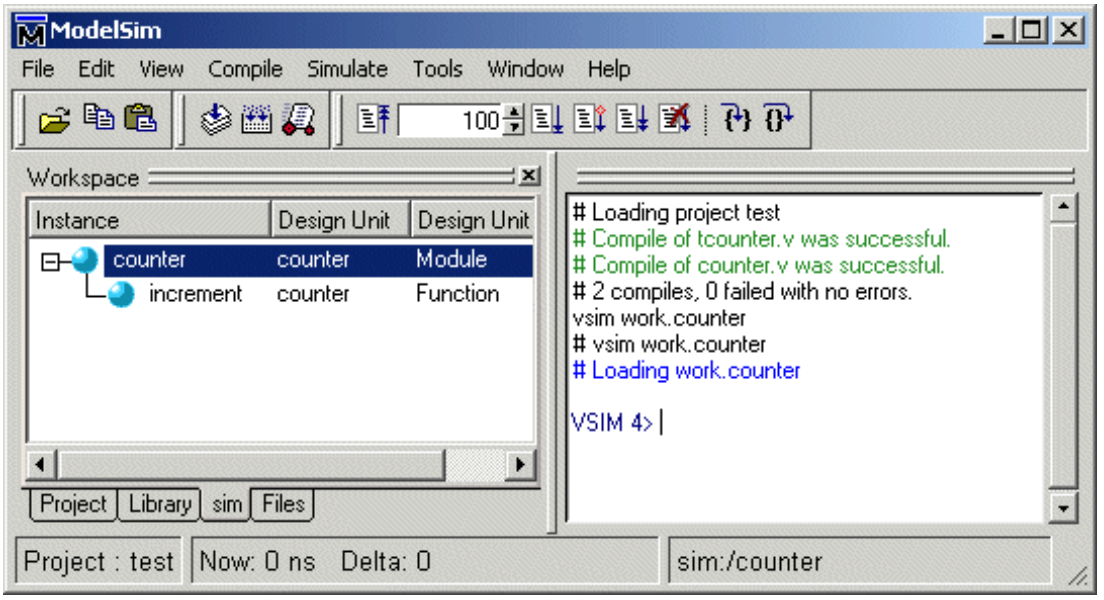


Once compilation is finished, click the Library tab, expand library *work* by clicking the "+", and you'll see the two compiled design units.



Step 4 — Simulating a design

To simulate one of the designs, either double-click the name or right-click the name and select Simulate. A new tab appears showing the structure of the active simulation.



At this point you are ready to run the simulation and analyze your results. You often do this by adding signals to the Wave window and running the simulation for a given period of time. See the *ModelSim Tutorial* for examples.

Other basic project operations

Open an existing project

If you previously exited ModelSim with a project open, ModelSim automatically will open that same project upon startup. You can open a different project by selecting **File** > **Open** > **Project** (Main window).

Close a project

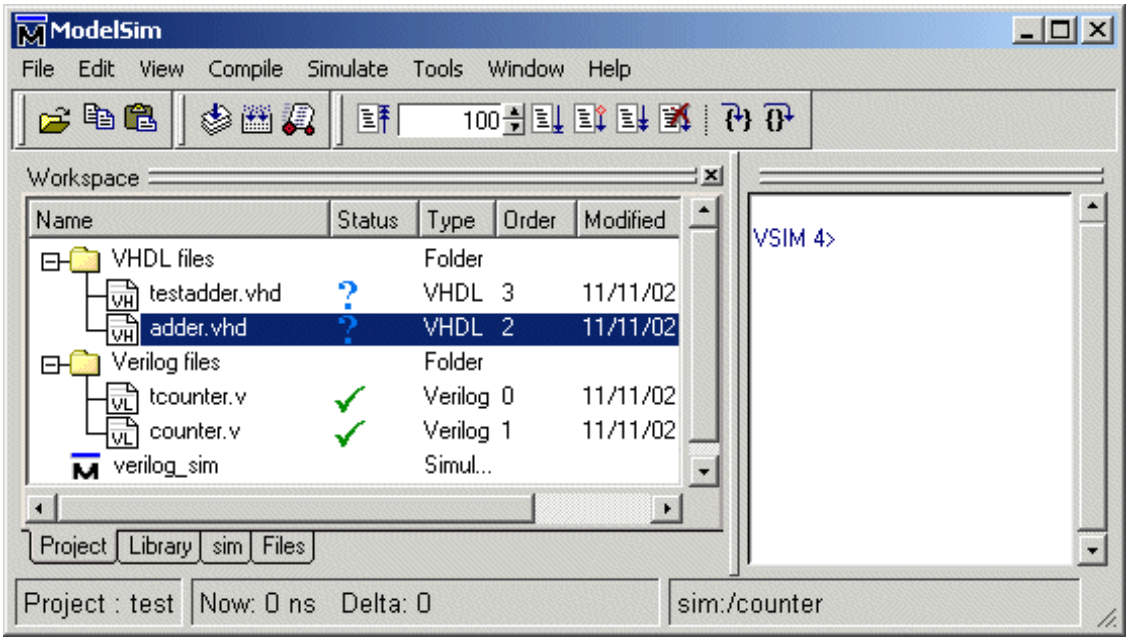
Select **File** > **Close** > **Project** (Main window). This closes the Project tab but leaves the Library tab open in the workspace. Note that you cannot close a project while a simulation is in progress.

Delete a project

Select **File** > **Delete** > **Project** (Main window). You cannot delete a project while it is open.

The Project tab

The Project tab contains information about the items in your project. By default the tab is divided into five columns.



Name – The name of a file or object.

Status – Identifies whether a source file has been successfully compiled. Applies only to VHDL or Verilog files. A question mark means the file hasn't been compiled or the source file has changed since the last successful compile; an X means the compile failed; a check mark means the compile succeeded.

Type – The file type as determined by registered file types on Windows or the type you specify when you add the file to the project.

Order – The order in which the file will be compiled when you execute a Compile All command.

Modified – The date and time of the last modification to the file.

You can hide or show columns by right-clicking on a column title and selecting or deselecting entries.

Sorting the list

You can sort the list by any of the five columns. Click on a column heading to sort by that column; click the heading again to invert the sort order. An arrow in the column heading indicates which field the list is sorted by and whether the sort order is descending (down arrow) or ascending (up arrow).

Project tab context menu

Like the other workspace tabs, the Project tab has a context menu that you access by clicking your right mouse button anywhere in the tab.

The context menu has the following options:

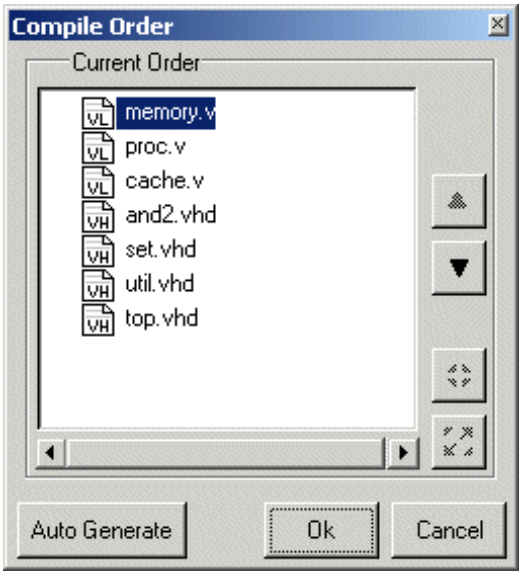
- **Edit**
  Open the selected file in the ModelSim editor.
- **Compile > Compile Selected**
  Compile the selected file(s). Note that if you select a folder and select **Compile Selected**, it will compile all files in the folder and any sub-folders.
- **Compile > Compile All**
  Compile all source files included in the project.
- **Compile > Compile Out-of-Date**
  Compile source files that have been modified since the last compile.
- **Compile > Compile Order**
  Set compile order for all files in the project. See "Changing compile order" (UM-28) for more details.
- **Compile > Compile Report**
  Show the compilation history of the selected file.
- **Compile > Compile Summary**
  Show the compilation history of the entire project.
- **Compile > Compile Properties**
  View/change project compiler settings for the selected source file(s).
- **Simulate**
  Load the design unit(s) and associated simulation options from the selected Simulation Configuration. See "Creating a Simulation Configuration" (UM-30) for more details.
- **Add to Project > New File**
  Add a new file to the project.
- **Add to Project > Existing File**
  Add an extant file to the project.
- **Add to Project > Simulation Configuration**
  Create a new Simulation Configuration. See "Creating a Simulation Configuration" (UM-30) for more details.
- **Add to Project > Folder**
  Add an organization folder to the project. See "Organizing projects with folders" (UM-32) for more details.
- **Remove from Project**
  Remove the selected item from the project.
- **Close Project**
  Close the active project.
- **Properties**
  View/change project compiler settings for the selected source file(s).

Changing compile order

When you compile all files in a project, ModelSim by default compiles the files in the order in which they were added to the project. You have two alternatives for changing the default compile order: 1) select and compile each file individually; 2) specify a custom compile order.

To specify a custom compile order, follow these steps:

1 Select **Compile > Compile Order** (Main window) or select it from the context menu in the Project tab.



2 Drag the files into the correct order or use the up and down arrow buttons. Note that you can select multiple files and drag them simultaneously.

Auto-generating compile order

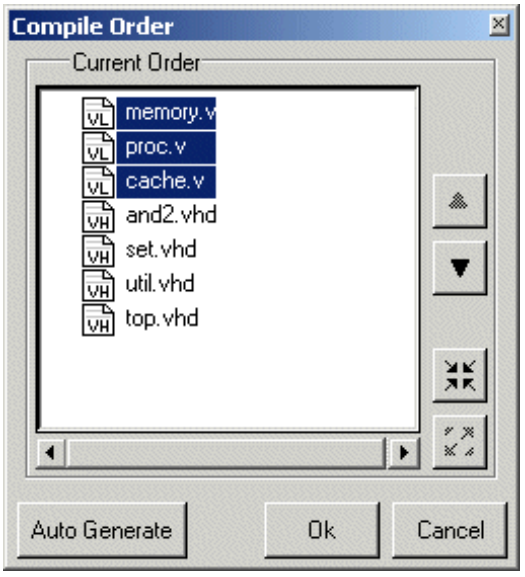
The **Auto Generate** button in the Compile Order dialog (see above) "determines" the correct compile order by making multiple passes over the files. It starts compiling from the top; if a file fails to compile due to dependencies, it moves that file to the bottom and then recompiles it after compiling the rest of the files. It continues in this manner until all files compile successfully or until a file(s) can't be compiled for reasons other than dependency.

Grouping files

You can group two or more files in the Compile Order dialog so they are sent to the compiler at the same time. For example, you might have one file with a bunch of Verilog define statements and a second file that is a Verilog module. You would want to compile these two files together.

To group files, follow these steps:

1 Select the files you want to group.



2 Click the Group button.

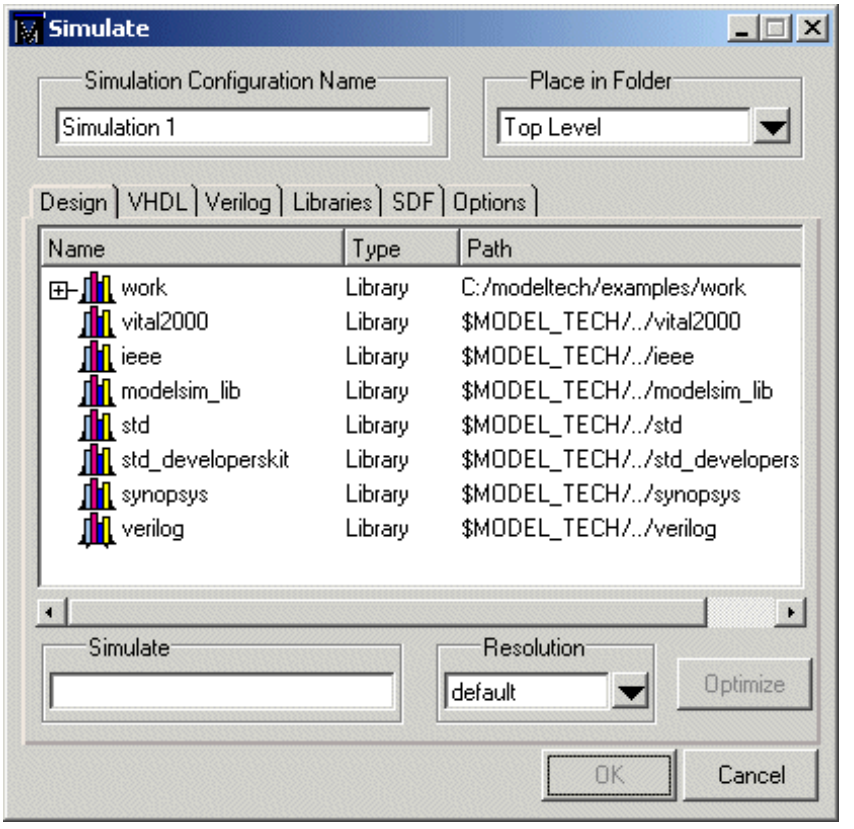
To ungroup files, select the group and click the Ungroup button.

Creating a Simulation Configuration

A Simulation Configuration associates a design unit(s) and its simulation options. For example, say you routinely load a particular design and you have to specify the simulator resolution, generics, and SDF timing files. Ordinarily you would have to specify those options each time you load the design. With a Simulation Configuration, you would specify the design and those options and then save the configuration with a name (e.g., *top_config*). The name is then listed in the Project tab and you can double-click it to load the design along with its options.

To create a Simulation Configuration, follow these steps:

1 Select **File > Add to Project > Simulation Configuration** (Main window) or select it from the context menu in the Project tab.



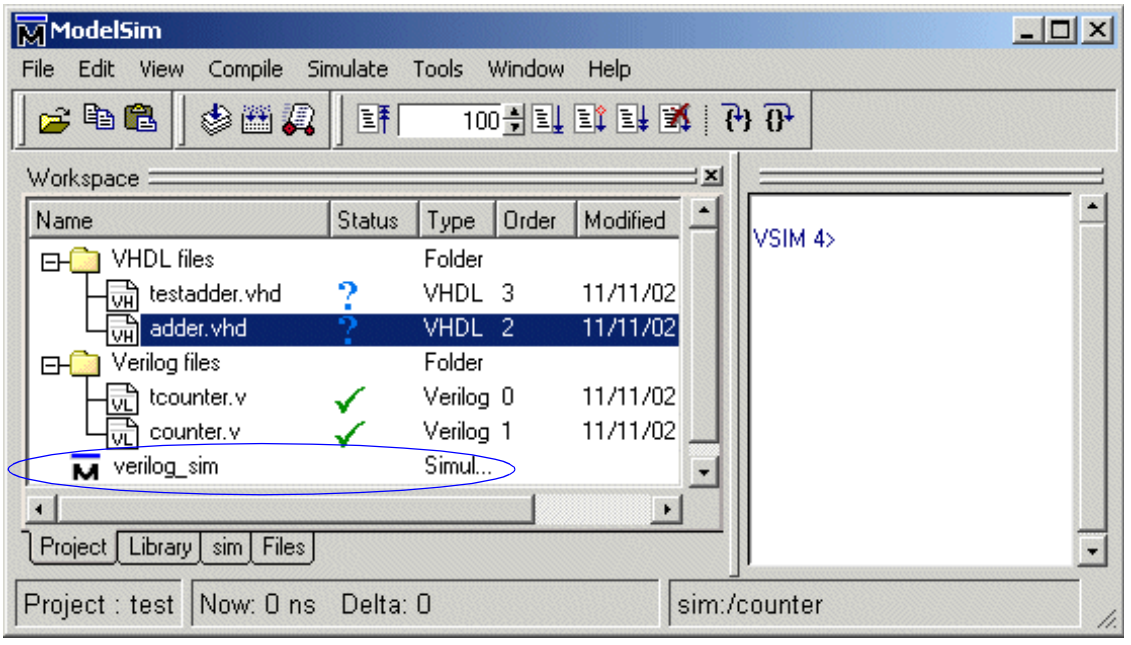
2 Specify a name in the **Simulation Configuration Name** field.

3 Specify the folder in which you want to place the configuration (see Organizing projects with folders (UM-32)).

4 Select one or more design unit(s) and click **Add**.

5 Use the other tabs in the dialog to specify any required simulation options. All of the options in this dialog are described under "Simulating with the graphic interface" (UM-245).

Click OK and the simulation configuration is added to the Project tab.



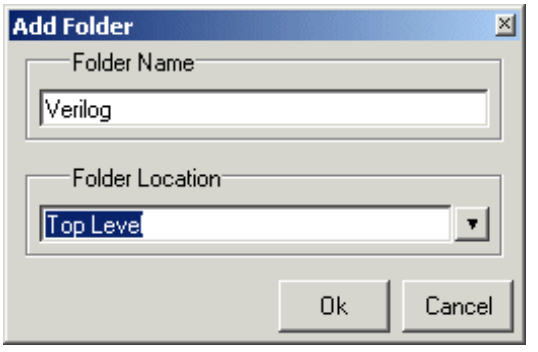
Double-click the object to load it.

Organizing projects with folders

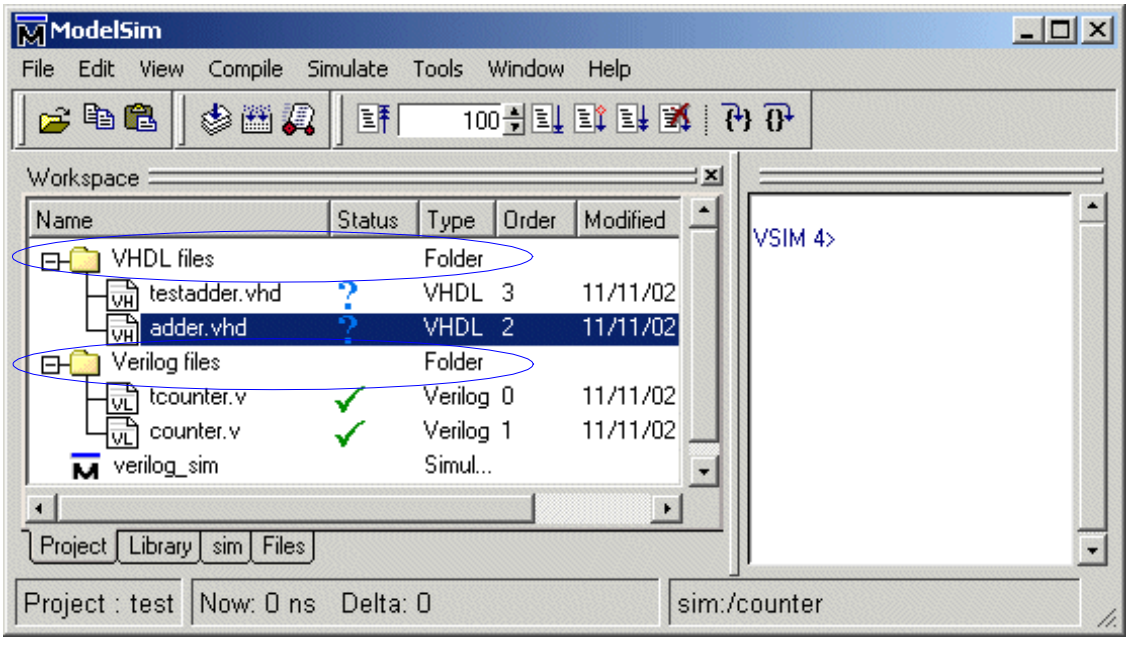
The more files you add to a project, the harder it can be to locate the item you need. You can add "folders" to the project to organize your files. These folders are akin to directories in that you can have multiple levels of folders and sub-folders. However, no actual directories are created via the file system–the folders are present only within the project file.

Adding a folder

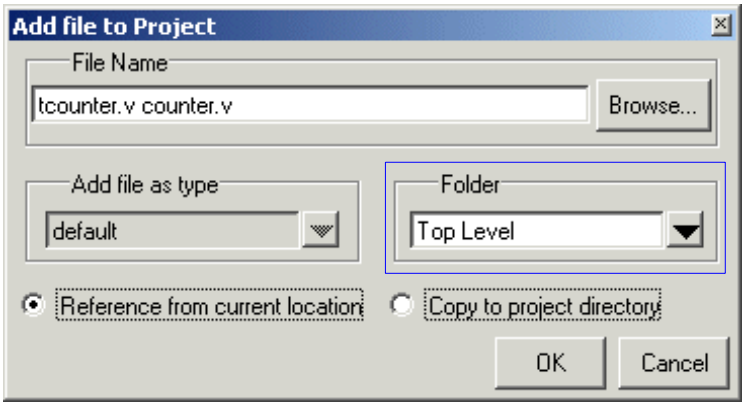
To add a folder to your project, select **File > Add to Project > Folder**.



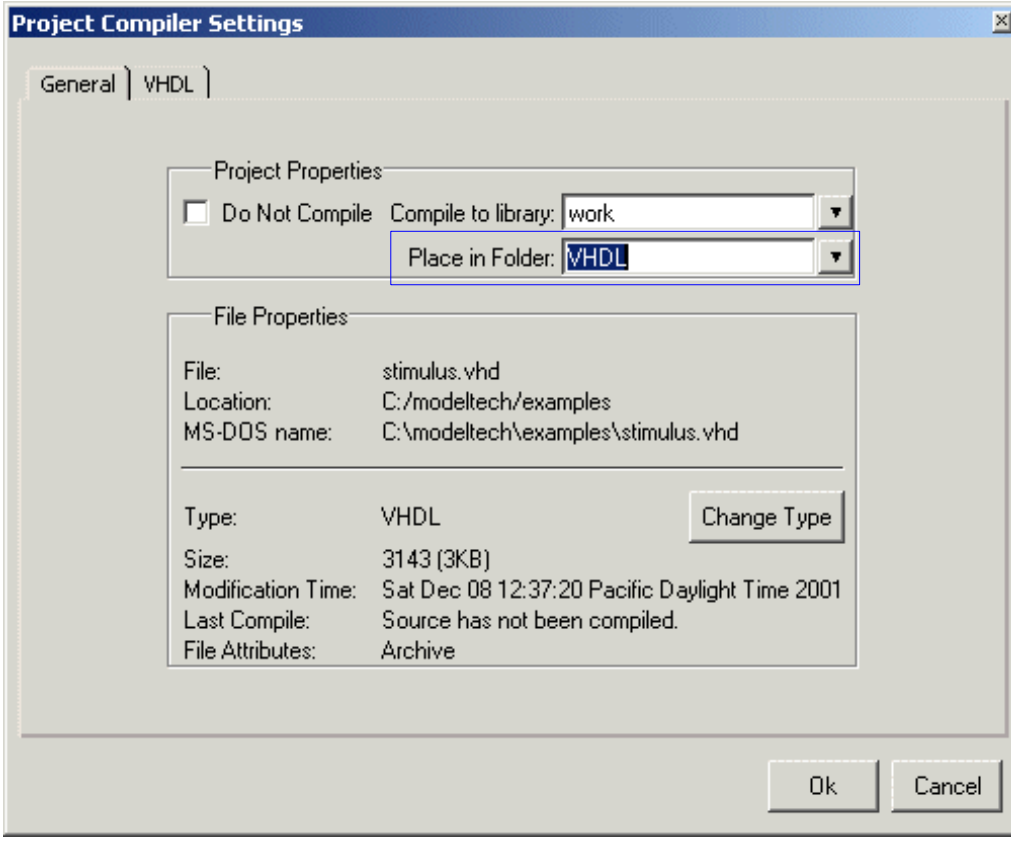
Specify the Folder Name, the location for the folder, and click OK. The folder will be displayed in the Project tab.



You use the folders when you add new objects to the project. For example, when you add a file, you can select which folder to place it in.



If you want to move a file into a folder later on, you can do so using the Properties dialog for the file (right-click on the file and select Properties from the context menu).

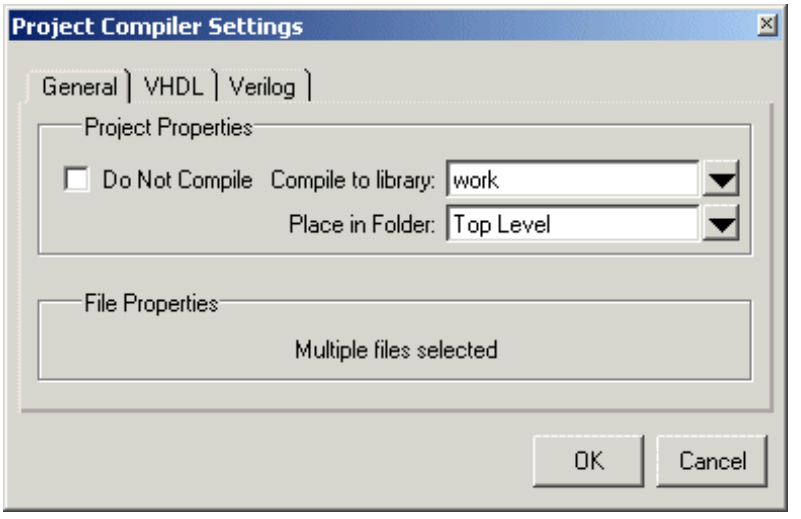


Setting compiler options

The VHDL and Verilog compilers (**vcom** and **vlog**, respectively) have numerous options that affect how a design is compiled and subsequently simulated. You can customize the settings on individual files or a group of files.

Important: Any changes you make to the compile properties outside of the project, whether from the command line, the GUI, or the *modelsim.ini* file, *will not* affect the properties of files already in the project.

To customize specific files, select the file(s) in the Project tab, right click on the file names, and select **Properties**. The resulting dialog varies depending on the number and type of files you have selected. If you select a single VHDL or Verilog file, you'll see the General tab and the VHDL or Verilog tab, respectively. On the General tab, you'll see file properties such as Type, Location, and Size. If you select multiple files, the file properties on the General tab are not listed. Finally, if you select both a VHDL file and a Verilog file, you'll see all three tabs but no file information on the General tab.



The General tab includes these options:

- **Do Not Compile**
  Determines whether the file is excluded from the compile.
- **Compile to library**
  Specifies to which library you want to compile the file; defaults to the working library.
- **Place in Folder**
  Specifies the folder in which to place the selected file(s). See "Organizing projects with folders" (UM-32) for details on folders.
- **File Properties**
  A variety of information about the selected file (e.g, type, size, path). Displays only if a single file is selected in the Project tab.

The definitions of the options on the VHDL and Verilog tabs can be found in the section "Setting default compile options" (UM-240).

When setting options on a group of files, keep in mind the following:

- If two or more files have different settings for the same option, the checkbox in the dialog will be "grayed out." If you change the option, you cannot change it back to a "multi- state setting" without cancelling out of the dialog. Once you click OK, ModelSim will set the option the same for all selected files.
- If you select a combination of VHDL and Verilog files, the options you set on the VHDL and Verilog tabs apply only to those file types.

Accessing projects from the command line

Generally, projects are used from within the ModelSim GUI. However, standalone tools will use the project file if they are invoked in the project's root directory. If you want to invoke outside the project directory, set the **MODELSIM** environment variable with the path to the project file (*<Project_Root_Dir>/<Project_Name>.mpf*).

You can also use the **project** command (CR-104) from the command line to perform common operations on new projects. The command is to be used outside of a simulation session.

3 - Design libraries

Chapter contents

Design library contents UM-38
Design unit information UM-38
Archives UM-38

Design library types UM-39

Working with design libraries UM-40
Creating a library UM-40
Managing library contents UM-41
Assigning a logical name to a design library UM-43
Moving a library UM-44

Specifying the resource libraries UM-45
VHDL resource libraries UM-45
Predefined libraries UM-46
Alternate IEEE libraries supplied UM-46
Regenerating your design libraries UM-47

Importing FPGA libraries UM-48

VHDL contains *libraries*, which are objects that contain compiled design units; libraries are given names so they may be referenced. Verilog designs simulated within ModelSim are compiled into libraries as well.

Design library contents

A *design library* is a directory or archive that serves as a repository for compiled design units. The design units contained in a design library consist of VHDL entities, packages, architectures, and configurations; and Verilog modules and UDPs (user-defined primitives). The design units are classified as follows:

- **Primary design units**
  Consist of entities, package declarations, configuration declarations, modules, and UDPs. Primary design units within a given library must have unique names.
- **Secondary design units**
  Consist of architecture bodies and package bodies. Secondary design units are associated with a primary design unit. Architectures by the same name can exist if they are associated with different entities.

Design unit information

The information stored for each design unit in a design library is:

- retargetable, executable code
- debugging information
- dependency information

Archives

By default design libraries are stored in a directory structure with a sub-directory for each design unit in the library. Alternatively, you can configure a design library to use archives. In this case each design unit is stored in its own archive file. To create an archive, use the **-archive** argument to the **vlib** command (CR-180).

Generally you would do this only in the rare case that you hit the reference count limit on I-nodes due to the ".." entries in the lower-level directories. An example of an error message that is produced when this limit is hit is:

```
mkdir: cannot create directory ‘65534’: Too many links
```

Archives may also have limited value to customers seeking disk space savings.

Note that GMAKE won’t work with these archives on the IBM platform.

Design library types

There are two kinds of design libraries: working libraries and resource libraries. A *working library* is the library into which a design unit is placed after compilation. A *resource library* contains design units that can be referenced within the design unit being compiled. Only one library can be the working library; in contrast, any number of libraries (including the working library itself) can be resource libraries during a compilation.

The library named **work** has special attributes within ModelSim; it is predefined in the compiler and need not be declared explicitly (i.e. **library work**). It is also the library name used by the compiler as the default destination of compiled design units. In other words the **work** library is the *working* library. In all other aspects it is the same as any other library.

Working with design libraries

The implementation of a design library is not defined within standard VHDL or Verilog. Within ModelSim design libraries are implemented as directories and can have any legal name allowed by the operating system, with one exception; extended identifiers are not supported for library names.

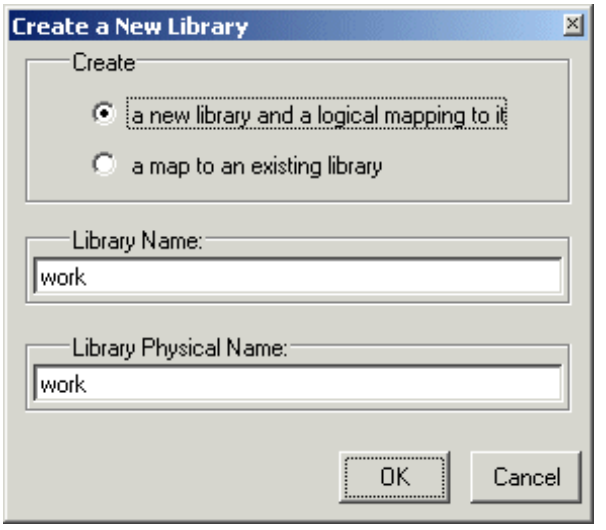
Creating a library

When you create a project (see "Getting started with projects" (UM-20)), ModelSim automatically creates a working design library. If you don't create a project, you need to create a working design library before you run the compiler. This can be done from either the command line or from the ModelSim graphic interface.

From the ModelSim prompt or a DOS prompt, use this **vlib** command (CR-180):

```
vlib <directory_pathname>
```

To create a new library with the ModelSim graphic interface, select **File > New > Library** (Main window).



The **Create a New Library** dialog box includes these options:

- **Create a new library and a logical mapping to it**
  Type the new library name into the **Library Name** field. This creates a library sub-directory in your current working directory, initially mapped to itself. Once created, the mapped library is easily remapped to a different library.
- **Create a map to an existing library**
  Type the new library name into the **Library Name** field, then type into the **Library Maps to** field or **Browse** to select a library name for the mapping.
- **Library Name**
  Type the logical name of the new library into this field.

- **Library Physical Name**
  Type the physical name of the new library into this field. ModelSim will create a directory with this name.
- **Library Maps to**
  Type or **Browse** for a mapping for the specified library. This field is visible and can be changed only when the **Create a map to an existing library** option is selected.

When you click **OK**, ModelSim creates the specified library directory and writes a specially-formatted file named *_info* into that directory. The *_info* file must remain in the directory to distinguish it as a ModelSim library.

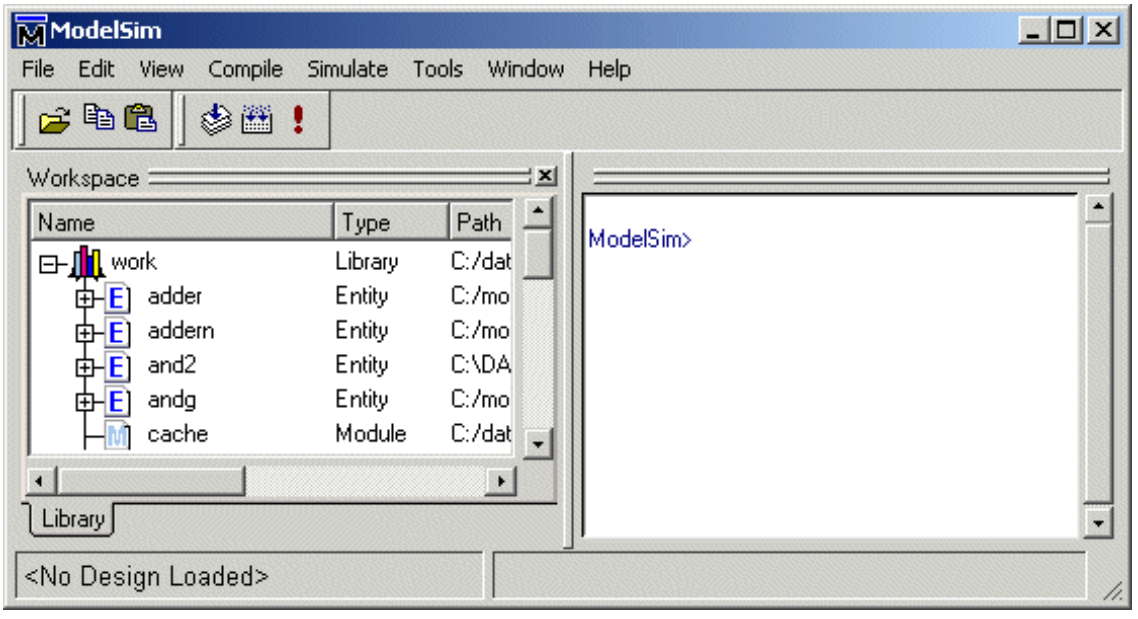
The new map entry is written to the *modelsim.ini* file in the [Library] section. See "[Library] library path variables" (UM-341) for more information.

Note: Remember that a design library is a special kind of directory; the only way to create a library is to use the ModelSim GUI or the **vlib** command (CR-180). Do not create libraries using DOS or Windows commands.

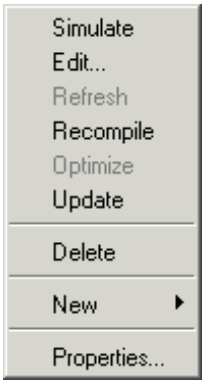
Managing library contents

Library contents can be viewed, deleted, recompiled, edited and so on using either the graphic interface or command line.

The Library tab in the Main window workspace provides access to design units (configurations, modules, packages, entities, and architectures) in a library. The listing is organized hierarchically, and the unit types are identified both by icon (entity (E), module (M), and so forth) and the Type column.



The Library tab has a context menu that you access by clicking your right mouse button in the Library tab.



The context menu includes the following commands:

- **Simulate**
  Loads the selected design unit and opens structure and Files tabs in the workspace. Related command line command is **vsim** (CR-189).
- **Edit**
  Opens the selected design unit in the Source window, or if a library is selected, opens the Edit Library Mapping dialog (see "Library mappings with the GUI" (UM-43)).
- **Refresh**
  Rebuilds the library image of the selected library without using source code. Related command line command is **vcom** (CR-145) or with the **-refresh** argument.
- **Recompile**
  Recompiles the selected design unit. Related command line command is **vcom** (CR-145) or .
- **Update**
  Updates the display of available libraries and design units.
- **Delete**
  Deletes the selected design unit. Related command line command is **vdel** (CR-151).

  Deleting a package, configuration, or entity will remove the design unit from the library. If you delete an entity that has one or more architectures, the entity and all its associated architectures will be deleted.

  You can also delete an architecture without deleting its associated entity. Expand the entity, right-click the desired architecture name, and select Delete. You are prompted for confirmation before any design unit is actually deleted.
- **New**
  Create a new library.
- **Properties**
  Displays various properties (e.g., Name, Type, Source, etc.) of the selected design unit or library.

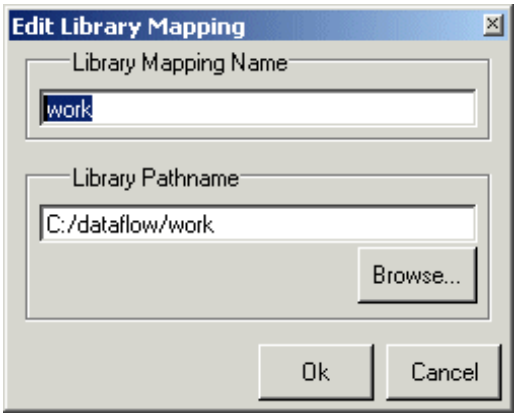
Assigning a logical name to a design library

VHDL uses logical library names that can be mapped to ModelSim library directories. By default, ModelSim can find libraries in your current directory (assuming they have the right name), but for it to find libraries located elsewhere, you need to map a logical library name to the pathname of the library.

You can use the GUI, a command, or a project to assign a logical name to a design library.

Library mappings with the GUI

To associate a logical name with a library, select the library in the workspace, right-click and select **Edit** from the context menu. This brings up a dialog box that allows you to edit the mapping.



The dialog box includes these options:

- **Library Mapping Name**
  The logical name of the library.
- **Library Pathname**
  The pathname to the library.

Library mapping from the command line

You can issue a command to set the mapping between a logical library name and a directory; its form is:

```
vmap <logical_name> <directory_pathname>
```

You may invoke this command from either a DOS prompt or from the command line within ModelSim.

When you use **vmap** (CR-188) this way you are modifying the *modelsim.ini* file. You can also modify *modelsim.ini* manually by adding a mapping line. To do this, use a text editor and add a line under the [Library] section heading using the syntax:

```
<logical_name> = <directory_pathname>
```

More than one logical name can be mapped to a single directory. For example, suppose the *modelsim.ini* file in the current working directory contains following lines:

```
[Library]
work = /usr/rick/design
my_asic = /usr/rick/design
```

This would allow you to use either the logical name **work** or **my_asic** in a **library** or **use** clause to refer to the same design library.

The **vmap** command (CR-188) can also be used to display the mapping of a logical library name to a directory. To do this, enter the shortened form of the command:

```
vmap <logical_name>
```

Library search rules

The system searches for the mapping of a logical name in the following order:

- First the system looks for a *modelsim.ini* file.
- If the system doesn't find a *modelsim.ini* file, or if the specified logical name does not exist in the *modelsim.ini* file, the system searches the current working directory for a subdirectory that matches the logical name.

An error is generated by the compiler if you specify a logical name that does not resolve to an existing directory.

Moving a library

Individual design units in a design library cannot be moved. An *entire* design library can be moved, however, by using standard operating system commands for moving a directory or an archive.

Specifying the resource libraries

Verilog resource libraries

ModelSim supports and encourages separate compilation of distinct portions of a Verilog design. The **vlog** (CR-181) compiler is used to compile one or more source files into a specified library. The library thus contains pre-compiled modules and UDPs that are referenced by the simulator as it loads the design. See "Library usage" (UM-72).

Important: Resource libraries are specified differently for Verilog and VHDL. For Verilog you use either the **-L** or **-Lf** argument to **vlog** (CR-181).

VHDL resource libraries

Within a VHDL source file, you use the VHDL **library** clause to specify logical names of one or more resource libraries to be referenced in the subsequent design unit. The scope of a **library** clause includes the text region that starts immediately after the **library** clause and extends to the end of the declarative region of the associated design unit. *It does not extend to the next design unit in the file.*

Note that the **library** clause is not used to specify the working library into which the design unit is placed after compilation; the **vcom** command (CR-145) adds compiled design units to the current working library. By default, this is the library named **work**. To change the current working library, you can use **vcom -work** and specify the name of the desired target library.

Default binding rules

A common question related to resource libraries is how ModelSim handles default binding for components. ModelSim addresses default binding at compile time. When looking for an entity to bind with, ModelSim searches the currently visible libraries for an entity with the same name as the component. ModelSim does this because IEEE 1076-1987 contained a flaw that made it almost impossible for an entity to be directly visible if it had the same name as the component. In short, if a component was declared in an architecture, any like-named entity above that declaration would be hidden because component/entity names cannot be overloaded. As a result we implemented the following rules for determining default binding:

- If a directly visible entity has the same name as the component, use it.
- If the component is declared in a package, search the library that contained the package for an entity with the same name.
- Search the work library.
- Search all other libraries that are currently visible by means of the **library** clause.

In IEEE 1076-1993, the flaw was partially fixed in that the name look-up for the default entity ignores component declarations. However, you could still encounter problems. Consider the case where you declare a component *C* in a package *P*, library *L* contains an entity *C*, and you have the following lines of code:

```
library L;
use L.P.all;  -- Makes component C visible
use L.all;      -- Because L.C exists and entity and component cannot be
                   overloaded, neither L.C nor L.P.C are directly visible.
```

In this case you couldn't have the statement:

```
U1: C  PORT MAP ( p1 => ...);
```

Instead, you need to have:

```
U1: P.C PORT MAP ( p1 => ...);
```

Because the default binding rules in IEEE 1076 contain these flaws, different simulators implement default binding in different ways.

Predefined libraries

Certain resource libraries are predefined in standard VHDL. The library named **std** contains the packages **standard** and **textio**, which should not be modified. The contents of these packages and other aspects of the predefined language environment are documented in the *IEEE Standard VHDL Language Reference Manual, Std 1076-1987* and *ANSI/IEEE Std 1076-1993*. See also, "Using the TextIO package" (UM-55).

A VHDL **use** clause can be specified to select particular declarations in a library or package that are to be visible within a design unit during compilation. A **use** clause references the compiled version of the package—not the source.

By default, every VHDL design unit is assumed to contain the following declarations:

```
LIBRARY std, work;
USE std.standard.all
```

To specify that all declarations in a library or package can be referenced, add the suffix *.all* to the library/package name. For example, the **use** clause above specifies that all declarations in the package *standard* in the design library named *std* are to be visible to the VHDL design file in which the **use** clause is placed. Other libraries or packages are not visible unless they are explicitly specified using a **library** or **use** clause.

Another predefined library is **work**, the library where a design unit is stored after it is compiled as described earlier. There is no limit to the number of libraries that can be referenced, but only one library is modified during compilation.

Alternate IEEE libraries supplied

The installation directory may contain two or more versions of the IEEE library:

- *ieeepure*
  Contains only IEEE approved std_logic_1164 packages (accelerated for ModelSim).

- *ieee*
  Contains precompiled Synopsys and IEEE arithmetic packages which have been accelerated by Model Technology including math_complex, math_real, numeric_bit, numeric_std, std_logic_1164, std_logic_misc, std_logic_textio, std_logic_arith, std_logic_signed, std_logic_unsigned, vital_primitives, and vital_timing.

You can select which library to use by changing the mapping in the *modelsim.ini* file. The *modelsim.ini* file in the installation directory defaults to the *ieee* library.

Regenerating your design libraries

Depending on your current ModelSim version, you may need to regenerate your design libraries before running a simulation. Check the installation README file to see if your libraries require an update. You can regenerate your design libraries using the **Refresh** command from the Library tab context menu (see "Managing library contents" (UM-41)), or by using the **-refresh** argument to **vcom** (CR-145) and **vlog** (CR-181).

From the command line, you would use vcom with the **-refresh** option to update VHDL design units in a library, and vlog with the **-refresh** option to update Verilog design units. By default, the work library is updated; use **-work <library>** to update a different library. For example, if you have a library named *mylib* that contains both VHDL and Verilog design units:

```
vcom -work mylib -refresh
vlog -work mylib -refresh
```

An important feature of **-refresh** is that it rebuilds the library image without using source code. This means that models delivered as compiled libraries without source code can be rebuilt for a specific release of ModelSim (4.6 and later only). In general, this works for moving forwards or backwards on a release. Moving backwards on a release may not work if the models used compiler switches or directives (Verilog only) that do not exist in the older release.

Note: You don't need to regenerate the *std*, *ieee*, *vital22b*, and *verilog* libraries. Also, you cannot use the **-refresh** option to update libraries that were built before the 4.6 release.

Importing FPGA libraries

ModelSim includes an import wizard for referencing and using vendor FPGA libraries. The wizard scans for and enforces dependencies in the libraries and determines the correct mappings and target directories.

Important: The FPGA libraries you import must be pre-compiled. Most FPGA vendors supply pre-compiled libraries configured for use with ModelSim.

To import an FPGA library, select **File > Import > Library** (Main window).

Import Library Wizard
The Import Library Wizard will step you through the tasks necessary to reference and use a library.
A library can be either an existing Model Technology library or an FPGA library that you received from an FPGA vendor. If the library was received from an FPGA vendor, it must be a precompiled library.
Please enter the location of the library to be imported below.
Import Library Pathname
Browse...
< Previous
Next >
Cancel

Follow the instructions in the wizard to complete the import.

4 - VHDL simulation

Chapter contents

Compiling VHDL designs UM-50
- Creating a design library UM-50
- Invoking the VHDL compiler UM-50
- Dependency checking UM-50
- Range and index checking UM-50

Simulating VHDL designs UM-52
- Simulator resolution limit UM-52
- Delta delays UM-53

Using the TextIO package UM-55
- Syntax for file declaration UM-55
- Using STD_INPUT and STD_OUTPUT within ModelSim . . . UM-56

TextIO implementation issues UM-57
- Writing strings and aggregates UM-57
- Reading and writing hexadecimal numbers UM-58
- Dangling pointers UM-58
- The ENDLINE function UM-58
- The ENDFILE function UM-58
- Using alternative input/output files UM-59
- Providing stimulus UM-59

VITAL specification and source code UM-60

VITAL packages UM-60

ModelSim VITAL compliance UM-60
- VITAL compliance checking UM-60
- Compiling and simulating with accelerated VITAL packages . . UM-61

Compiling and simulating with accelerated VITAL packages . . . UM-61

Util package UM-62
- get_resolution UM-62
- init_signal_driver() UM-63
- init_signal_spy() UM-63
- signal_force() UM-63
- signal_release() UM-63
- to_real() UM-64
- to_time() UM-65

This chapter provides an overview of compilation and simulation for VHDL; using the TextIO package with ModelSim; ModelSim's implementation of the VITAL (VHDL Initiative Towards ASIC Libraries) specification for ASIC modeling; and documentation on ModelSim's special built-in utilities package.

The TextIO package is defined within the *VHDL Language Reference Manuals, IEEE Std 1076-1987* and *IEEE Std 1076-1993*; it allows human-readable text input from a declared source within a VHDL file during simulation.

Compiling VHDL designs

Creating a design library

Before you can compile your design, you must create a library in which to store the compilation results. Use **vlib** (CR-180) to create a new library. For example:

```
vlib work
```

This creates a library named **work**. By default, compilation results are stored in the **work** library.

Note: The **work** library is actually a subdirectory named *work*. This subdirectory contains a special file named *_info*. Do not create libraries using MS Windows or DOS commands – always use the **vlib** command (CR-180).

See "Design libraries" (UM-37) for additional information on working with libraries.

Invoking the VHDL compiler

ModelSim compiles one or more VHDL design units with a single invocation of **vcom** (CR-145), the VHDL compiler. The design units are compiled in the order that they appear on the command line. For VHDL, the order of compilation is important – you must compile any entities or configurations before an architecture that references them.

You can simulate a design containing units written with both the 1076 -1987 and 1076 -1993 versions of VHDL. To do so you will need to compile units from each VHDL version separately. The **vcom** (CR-145) command compiles units written with version 1076 -1987 by default; use the **-93** option with **vcom** (CR-145) to compile units written with version 1076 -1993. You can also change the default by modifying the *modelsim.ini* file (see "Preference variables located in INI files" (UM-341) for more information).

Dependency checking

Dependent design units must be reanalyzed when the design units they depend on are changed in the library. **vcom** (CR-145) determines whether or not the compilation results have changed. For example, if you keep an entity and its architectures in the same source file and you modify only an architecture and recompile the source file, the entity compilation results will remain unchanged and you will not have to recompile design units that depend on the entity.

Range and index checking

A range check verifies that a scalar value defined with a range subtype is always assigned a value within its range. An index check verifies that whenever an array subscript expression is evaluated, the subscript will be within the array's range.

Range and index checks are performed by default when you compile your design. You can disable range checks (potentially offering a performance advantage) and index checks using arguments to the **vcom** (CR-145) command. Or, you can use the **NoRangeCheck** and **NoIndexCheck** variables in the *modelsim.ini* file to specify whether or not they are performed. See "Preference variables located in INI files" (UM-341).

Range checks in ModelSim are slightly more restrictive than those specified by the VHDL LRM. ModelSim requires any assignment to a signal to also be in range whereas the LRM requires only that range checks be done whenever a signal is updated. Most assignments to signals update the signal anyway, and the more restrictive requirement allows ModelSim to generate better error messages.

Simulating VHDL designs

After compiling the design units, you can simulate your designs with **vsim** (CR-189). This section discusses simulation from the Windows/DOScommand line. You can also use a project to simulate (see "Getting started with projects" (UM-20)) or the **Simulate** dialog box (see "Simulating with the graphic interface" (UM-245)).

For VHDL invoke **vsim** (CR-189) with the name of the configuration, or entity/architecture pair. Note that if you specify a configuration you may not specify an architecture.

This example invokes **vsim** (CR-189) on the entity **my_asic** and the architecture **structure**:

```
vsim my_asic structure
```

vsim (CR-189) is capable of annotating a design using VITAL compliant models with timing data from an SDF file. You can specify the min:typ:max delay by invoking **vsim** with the **-sdfmin**, **-sdftyp** and **-sdfmax** options. Using the SDF file *f1.sdf* in the current work directory, the following invocation of **vsim** annotates maximum timing values for the design unit *my_asic*:

```
vsim -sdfmax /my_asic=f1.sdf my_asic
```

By default, the timing checks within VITAL models are enabled. They can be disabled with the **+notimingchecks** option. For example:

```
vsim +notimingchecks topmod
```

Simulator resolution limit

The simulator internally represents time as a 64-bit integer in units equivalent to the smallest unit of simulation time, also known as the simulator resolution limit. The default resolution limit is set to the value specified by the **Resolution** (UM-347) variable in the *modelsim.ini* file. You can view the current resolution by invoking the **report** command (CR-109) with the **simulator state** option.

Overriding the resolution

You can override ModelSim's default resolution by specifying the **-t** option on the command line or by selecting a different Simulator Resolution in the **Simulate** dialog box. Available resolutions are: 1x, 10x or 100x of fs, ps, ns, us, ms, or sec.

For example this command chooses 10 ps resolution:

```
vsim -t 10ps topmod
```

Clearly you need to be careful when doing this type of operation. If the resolution set by **-t** is larger than a delay value in your design, the delay values in that design unit are rounded to the next multiple of the resolution. In the example above, a delay of 4 ps would be rounded to 0 ps.

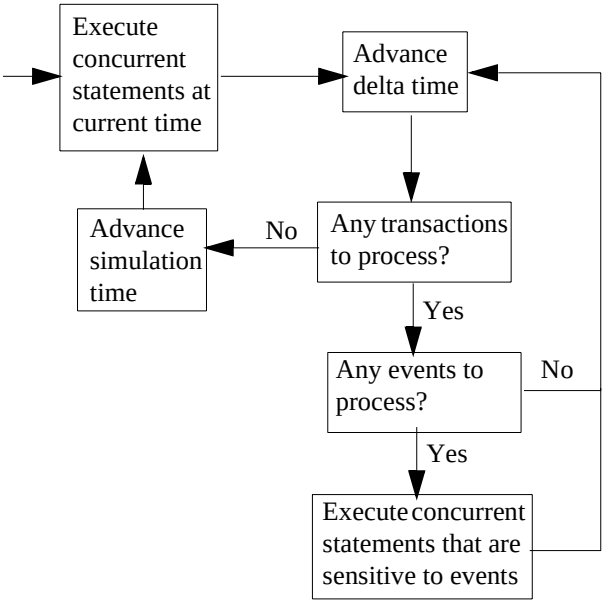
Choosing the resolution

You should choose the coarsest resolution limit possible that does not result in undesired rounding of your delays. The time precision should not be unnecessarily small because it will limit the maximum simulation time limit, and it will degrade performance in some cases.

Delta delays

Event-based simulators such as ModelSim may process many events at a given simulation time. Multiple signals may need updating, statements that are sensitive to these signals must be executed, and any new events that result from these statements must then be queued and executed as well. The steps taken to evaluate the design without advancing simulation time are referred to as "delta times" or just "deltas."

The diagram below represents the process for VHDL designs. This process continues until the end of simulation time.



This mechanism in event-based simulators may cause unexpected results. Consider the following code snippet:

```
.
.
.
  clk2 <= clk;

  process (rst, clk)
  begin
    if(rst = '0')then
      s0 <= '0';
    elsif(clk'event and clk='1') then
      s0 <= inp;

    end if;
  end process;


process (rst, clk2)
  begin
    if(rst = '0')then
```

```
      s1 <= '0';
    elsif(clk2'event and clk2='1') then
      s1 <= s0;
    end if;
  end process;
.
.
.
```

In this example you have two synchronous processes, one triggered with *clk* and the other with *clk2*. To your surprise, the signals change in the *clk2* process on the same edge as they are set in the *clk* process. As a result, the value of *inp* appears at *s1* rather than *s0*. What is going on?

Here is what's happing. During simulation an event on *clk* occurs (from the testbench). From this event ModelSim performs the "clk2 <= clk" assignment and the process which is sensitive to *clk*. Before advancing the simulation time, ModelSim finds that the process sensitive to *clk2* can also be run. Since there are no delays present, the effect is that the value of *inp* appears at *s1* in the same simulation cycle.

In order to get the expected results, you must do one of the following:

1 insert delay at every output

2 make certain to use the same clock

3 insert a delta delay

To insert a delta delay, you would modify the code like this:

```
process (rst, clk)
  begin
    if(rst = '0')then
      s0 <= '0';
    elsif(clk'event and clk='1') then
      s0 <= inp;
      s0_delayed <= s0;
    end if;
  end process;

 process (rst, clk2)
  begin
    if(rst = '0')then
      s1 <= '0';
    elsif(clk2'event and clk2='1') then
      s1 <= s0_delayed;
    end if;
  end process;
```

The best way to debug delta delay problems is observe your signals in the List window. There you can see how values change at each delta time.

Using the TextIO package

To access the routines in TextIO, include the following statement in your VHDL source code:

```
USE std.textio.all;
```

A simple example using the package TextIO is:

```
USE std.textio.all;
ENTITY simple_textio IS
END;

ARCHITECTURE simple_behavior OF simple_textio IS
BEGIN
    PROCESS
        VARIABLE i: INTEGER:= 42;
        VARIABLE LLL: LINE;
    BEGIN
        WRITE (LLL, i);
        WRITELINE (OUTPUT, LLL);
        WAIT;
    END PROCESS;
END simple_behavior;
```

Syntax for file declaration

The VHDL'87 syntax for a file declaration is:

```
file identifier : subtype_indication is [ mode ]  file_logical_name ;
```

where "file_logical_name" must be a string expression.

The VHDL'93 syntax for a file declaration is:

```
file identifier_list : subtype_indication [ file_open_information ] ;
```

where "`file_open_information`" is:

```
[open file_open_kind_expression] is file_logical_name
```

You can specify a full or relative path as the file_logical_name; for example (VHDL'87):

Normally if a file is declared within an architecture, process, or package, the file is opened when you start the simulator and is closed when you exit from it. If a file is declared in a subprogram, the file is opened when the subprogram is called and closed when execution RETURNs from the subprogram. Alternatively, the opening of files can be delayed until the first read or write by setting the **DelayFileOpen** variable in the *modelsim.ini* file. Also, the number of concurrently open files can be controlled by the **ConcurrentFileLimit** variable. These variables help you manage a large number of files during simulation. See *Appendix A - ModelSim variables* for more details.

Using STD_INPUT and STD_OUTPUT within ModelSim

The standard VHDL'87 TextIO package contains the following file declarations:

```
file input: TEXT is in "STD_INPUT";
file output: TEXT is out "STD_OUTPUT";
```

The standard VHDL'93 TextIO package contains these file declarations:

```
file input: TEXT open read_mode is "STD_INPUT";
file output: TEXT open write_mode is "STD_OUTPUT";
```

STD_INPUT is a file_logical_name that refers to characters that are entered interactively from the keyboard, and STD_OUTPUT refers to text that is displayed on the screen.

In ModelSim, reading from the STD_INPUT file allows you to enter text into the current buffer from a prompt in the Main window. The lines written to the STD_OUTPUT file appear in the Main window transcript.

TextIO implementation issues

Writing strings and aggregates

A common error in VHDL source code occurs when a call to a WRITE procedure does not specify whether the argument is of type STRING or BIT_VECTOR. For example, the VHDL procedure:

```
WRITE (L, "hello");
```

will cause the following error:

```
ERROR: Subprogram "WRITE" is ambiguous.
```

In the TextIO package, the WRITE procedure is overloaded for the types STRING and BIT_VECTOR. These lines are reproduced here:

```
procedure WRITE(L: inout LINE; VALUE: in BIT_VECTOR;
    JUSTIFIED: in SIDE:= RIGHT; FIELD: in WIDTH := 0);

procedure WRITE(L: inout LINE; VALUE: in STRING;
    JUSTIFIED: in SIDE:= RIGHT; FIELD: in WIDTH := 0);
```

The error occurs because the argument "hello" could be interpreted as a string or a bit vector, but the compiler is not allowed to determine the argument type until it knows which function is being called.

The following procedure call also generates an error:

```
WRITE (L, "010101");
```

This call is even more ambiguous, because the compiler could not determine, even if allowed to, whether the argument "010101" should be interpreted as a string or a bit vector.

There are two possible solutions to this problem:

- Use a qualified expression to specify the type, as in:

```
WRITE (L, string'("hello"));
```

- Call a procedure that is not overloaded, as in:

```
WRITE_STRING (L, "hello");
```

The WRITE_STRING procedure simply defines the value to be a STRING and calls the WRITE procedure, but it serves as a shell around the WRITE procedure that solves the overloading problem. For further details, refer to the WRITE_STRING procedure in the io_utils package, which is located in the file *<install_dir>/modeltech/examples/io_utils.vhd*.

Reading and writing hexadecimal numbers

The reading and writing of hexadecimal numbers is not specified in standard VHDL. The Issues Screening and Analysis Committee of the VHDL Analysis and Standardization Group (ISAC-VASG) has specified that the TextIO package reads and writes only decimal numbers.

To expand this functionality, ModelSim supplies hexadecimal routines in the package io_utils, which is located in the file *<install_dir>/modeltech/examples/io_utils.vhd*. To use these routines, compile the io_utils package and then include the following use clauses in your VHDL source code:

```
use std.textio.all;
use work.io_utils.all;
```

Dangling pointers

Dangling pointers are easily created when using the TextIO package, because WRITELINE de-allocates the access type (pointer) that is passed to it. Following are examples of good and bad VHDL coding styles:

Bad VHDL (because L1 and L2 both point to the same buffer):

```
READLINE (infile, L1);    -- Read and allocate buffer
L2 := L1;                 -- Copy pointers
WRITELINE (outfile, L1);  -- Deallocate buffer
```

Good VHDL (because L1 and L2 point to different buffers):

```
READLINE (infile, L1);      -- Read and allocate buffer
L2 := new string'(L1.all);  -- Copy contents
WRITELINE (outfile, L1);    -- Deallocate buffer
```

The ENDLINE function

The ENDLINE function described in the *IEEE Standard VHDL Language Reference Manual*, *IEEE Std 1076-1987* contains invalid VHDL syntax and cannot be implemented in VHDL. This is because access types must be passed as variables, but functions only allow constant parameters.

Based on an ISAC-VASG recommendation the ENDLINE function has been removed from the TextIO package. The following test may be substituted for this function:

```
(L = NULL) OR (L'LENGTH = 0)
```

The ENDFILE function

In the *VHDL Language Reference Manuals, IEEE Std 1076-1987 and IEEE Std 1076-1993*, the ENDFILE function is listed as:

```
-- function ENDFILE (L: in TEXT) return BOOLEAN;
```

As you can see, this function is commented out of the standard TextIO package. This is because the ENDFILE function is implicitly declared, so it can be used with files of any type, not just files of type TEXT.

Using alternative input/output files

You can use the TextIO package to read and write to your own files. To do this, just declare an input or output file of type TEXT. For example, for an input file:

The VHDL'87 declaration is:

```
file myinput : TEXT is in "pathname.dat";
```

The VHDL'93 declaration is:

```
file myinput : TEXT open read_mode is "pathname.dat";
```

Then include the identifier for this file ("myinput" in this example) in the READLINE or WRITELINE procedure call.

Providing stimulus

You can stimulate and test a design by reading vectors from a file, using them to drive values onto signals, and testing the results. A VHDL test bench has been included with the ModelSim install files as an example. Check for this file:

<*install_dir*>*/modeltech/examples/stimulus.vhd*

VITAL specification and source code

VITAL ASIC Modeling Specification

The IEEE 1076.4 VITAL ASIC Modeling Specification is available from the Institute of Electrical and Electronics Engineers, Inc.:

IEEE Customer Service
445 Hoes Lane
Piscataway, NJ 08855-1331

Tel: (732) 981-0060
Fax: (732) 981-1721
home page: http://www.ieee.org

VITAL source code

The source code for VITAL packages is provided in the */<install_dir>/vhdl_src/vital22b, /vital95, or /vital2000* directories.

VITAL packages

VITAL 1995 accelerated packages are pre-compiled into the **ieee** library in the installation directory. VITAL 2000 accelerated packages are pre-compiled into the **vital2000** library. If you need to use the newer library, you'll need to add a **use** clause to your VHDL code to access the VITAL 2000 packages. For example:

```
LIBRARY vital2000;
USE vital2000.all
```

ModelSim VITAL compliance

A simulator is VITAL compliant if it implements the SDF mapping and if it correctly simulates designs using the VITAL packages, as outlined in the VITAL Model Development Specification. ModelSim is compliant with the IEEE 1076.4 VITAL ASIC Modeling Specification. In addition, ModelSim accelerates the VITAL_Timing, VITAL_Primitives, and VITAL_memory packages. The optimized procedures are functionally equivalent to the IEEE 1076.4 VITAL ASIC Modeling Specification (VITAL 1995 and 2000).

VITAL compliance checking

If you are using VITAL 2.2b, you must turn off the compliance checking either by not setting the attributes, or by invoking **vcom** (CR-145) with the option **-novitalcheck**.

Compiling and simulating with accelerated VITAL packages

vcom (CR-145) automatically recognizes that a VITAL function is being referenced from the **ieee** library and generates code to call the optimized built-in routines.

Invoke with the **-novital** option if you do not want to use the built-in VITAL routines (when debugging for instance). To exclude all VITAL functions, use **-novital all**:

```
vcom -novital all design.vhd
```

To exclude selected VITAL functions, use one or more **-novital <fname>** options:

```
vcom -novital VitalTimingCheck -novital VitalAND design.vhd
```

The **-novital** switch only affects calls to VITAL functions from the design units currently being compiled. Pre-compiled design units referenced from the current design units will still call the built-in functions unless they too are compiled with the **-novital** option.

ModelSim VITAL built-ins will be updated in step with new releases of the VITAL packages.

Util package

The util package, included in ModelSim versions 5.5 and later, serves as a container for various VHDL utilities. The package is part of the modelsim_lib library which is located in the modeltech tree and is mapped in the default *modelsim.ini* file.

To access the utilities in the package, you would add lines like the following to your VHDL code:

```
library modelsim_lib;
use modelsim_lib.util.all;
```

get_resolution

get_resolution returns the current simulator resolution as a real number. For example, 1 femtosecond corresponds to 1e-15.

Syntax

```
resval := get_resolution;
```

Returns

Name	Type	Description
resval	real	The simulator resolution represented as a real

Arguments

None

Related functions

to_real() (UM-64)

to_time() (UM-65)

Example

If the simulator resolution is set to 10ps, and you invoke the command:

```
resval := get_resolution;
```

the value returned to resval would be 1e-11.

init_signal_driver()

The init_signal_driver() procedure drives the value of a VHDL signal or Verilog net onto an existing VHDL signal or Verilog net. This allows you to drive signals or nets at any level of the design hierarchy from within a VHDL architecture (e.g., a testbench).

See init_signal_driver (UM-271) in *Chapter 8 - Signal Spy* for complete details and syntax on this procedure.

init_signal_spy()

The init_signal_spy() utility mirrors the value of a VHDL signal or Verilog register/net onto an existing VHDL signal or Verilog register. This allows you to reference signals, registers, or nets at any level of hierarchy from within a VHDL architecture (e.g., a testbench).

See init_signal_spy (UM-274) in *Chapter 8 - Signal Spy* for complete details and syntax on this procedure.

signal_force()

The signal_force() procedure forces the value specified onto an existing VHDL signal or Verilog register or net. This allows you to force signals, registers, or nets at any level of the design hierarchy from within a VHDL architecture (e.g., a testbench). A signal_force works the same as the **force** command (CR-82) with the exception that you cannot issue a repeating force.

See signal_force (UM-276) in *Chapter 8 - Signal Spy* for complete details and syntax on this procedure.

signal_release()

The signal_release() procedure releases any force that was applied to an existing VHDL signal or Verilog register or net. This allows you to release signals, registers, or nets at any level of the design hierarchy from within a VHDL architecture (e.g., a testbench). A signal_release works the same as the **noforce** command (CR-92).

See signal_release (UM-278) in *Chapter 8 - Signal Spy* for complete details and syntax on this procedure.

to_real()

to_real() converts the physical type time value into a real value with respect to the current simulator resolution. The precision of the converted value is determined by the simulator resolution. For example, if you were converting 1900 fs to a real and the simulator resolution was ps, then the real value would be 2.0 (i.e., 2 ps).

Syntax

```
realval := to_real(timeval);
```

Returns

Name	Type	Description
realval	real	The time value represented as a real with respect to the simulator resolution

Arguments

Name	Type	Description
timeval	time	The value of the physical type time

Related functions

get_resolution (UM-62)

to_time() (UM-65)

Example

If the simulator resolution is set to ps, and you enter the following function:

```
realval := to_real(12.99 ns);
```

then the value returned to realval would be 12990.0. If you wanted the returned value to be in units of nanoseconds (ns) instead, you would use the **get_resolution** (UM-62) function to recalculate the value:

```
realval := 1e+9 * (to_real(12.99 ns)) * get_resolution();
```

If you wanted the returned value to be in units of femtoseconds (fs), you would enter the function this way:

```
realval := 1e+15 * (to_real(12.99 ns)) * get_resolution();
```

to_time()

to_time() converts a real value into a time value with respect to the current simulator resolution. The precision of the converted value is determined by the simulator resolution. For example, if you were converting 5.9 to a time and the simulator resolution was ps, then the time value would be 6 ps.

Syntax

```
timeval := to_time(realval);
```

Returns

Name	Type	Description
timeval	time	The real value represented as a physical type time with respect to the simulator resolution

Arguments

Name	Type	Description
realval	real	The value of the type real

Related functions

get_resolution (UM-62)

to_real() (UM-64)

Example

If the simulator resolution is set to ps, and you enter the following function:

```
timeval := to_time(72.49);
```

then the value returned to timeval would be 72 ps.

5 - Verilog simulation

Chapter contents

Compilation UM-69
- Incremental compilation UM-70
- Library usage UM-72
- Verilog-XL compatible compiler arguments UM-73
- Verilog-XL ‘uselib compiler directive UM-74

Simulation UM-76
- Invoking the simulator UM-76
- Simulator resolution limit UM-77
- Event ordering in Verilog designs UM-79
- Negative timing check limits UM-83
- Verilog-XL compatible simulator arguments UM-86

Cell libraries UM-87
- SDF timing annotation UM-87
- Delay modes UM-87

System tasks UM-89
- IEEE Std 1364 system tasks UM-89
- Verilog-XL compatible system tasks UM-92
- ModelSim Verilog system tasks UM-94

Compiler directives UM-95
- IEEE Std 1364 compiler directives UM-95
- Verilog-XL compatible compiler directives UM-96

Verilog PLI/VPI UM-97
- Registering PLI applications UM-97
- Registering VPI applications UM-99
- Compiling and linking PLI/VPI C applications UM-101
- Compiling and linking PLI/VPI C++ applications UM-102
- Specifying the PLI/VPI file to load UM-103
- PLI example UM-104
- VPI example UM-105
- The PLI callback reason argument UM-106
- The sizetf callback function UM-107
- PLI object handles UM-107
- Third party PLI applications UM-108
- Support for VHDL objects UM-109
- IEEE Std 1364 ACC routines UM-110
- IEEE Std 1364 TF routines UM-111
- Verilog-XL compatible routines UM-113
- 64-bit support in the PLI UM-113
- PLI/VPI tracing UM-113
- Debugging PLI/VPI application code UM-115

This chapter describes how to compile and simulate Verilog designs with ModelSim Verilog. ModelSim Verilog implements the Verilog language as defined by the IEEE Std 1364, and it is recommended that you obtain this specification as a reference manual.

In addition to the functionality described in the IEEE Std 1364, ModelSim Verilog includes the following features:

- Standard Delay Format (SDF) annotator compatible with many ASIC and FPGA vendor's Verilog libraries
- Value Change Dump (VCD) file extensions for ASIC vendor test tools
- Dynamic loading of PLI/VPI applications
- Compilation into retargetable, executable code
- Incremental design compilation
- Extensive support for mixing VHDL and Verilog in the same design (including SDF annotation)
- Graphic Interface that is common with ModelSim VHDL
- Extensions to provide compatibility with Verilog-XL

The following IEEE Std 1364 functionality is partially implemented in ModelSim Verilog:

- Verilog Procedural Interface (VPI) (see */<install_dir>/modeltech/docs/technotes/Verilog_VPI.note* for details)
- Verilog 2001 (see */<install_dir>/modeltech/docs/technotes/vlog_2000.note* for details)

Many of the examples in this chapter are shown from the command line. For compiling and simulating within a project or ModelSim's GUI see:

- Getting started with projects (UM-20)
- Compiling with the graphic interface (UM-238)
- Simulating with the graphic interface (UM-245)

Compilation

Before you can simulate a Verilog design, you must first create a library and compile the Verilog source code into that library. This section provides detailed information on compiling Verilog designs. For information on creating a design library, see *Chapter 3 - Design libraries*.

The ModelSim Verilog compiler, **vlog**, compiles Verilog source code into retargetable, executable code, meaning that the library format is compatible across all supported platforms and that you can simulate your design on any platform without having to recompile your design specifically for that platform. As you compile your design, the resulting object code for modules and UDPs is generated into a library. By default, the compiler places results into the work library. You can specify an alternate library with the **-work** argument. The following is a simple example of how to create a work library, compile a design, and simulate it:

Contents of top.v:

```
module top;
    initial $display("Hello world");
endmodule
```

Create the work library:

```
% vlib work
```

Compile the design:

```
% vlog top.v
-- Compiling module top

Top level modules:
    top
```

View the contents of the work library (optional):

```
% vdir
MODULE top
```

Simulate the design:

```
% vsim -c top
# Loading work.top
VSIM 1> run -all
# Hello world
VSIM 2> quit
```

In this example, the simulator was run without the graphic interface by specifying the **-c** argument. After the design was loaded, the simulator command **run -all** was entered, meaning to simulate until there are no more simulator events. Finally, the quit command was entered to exit the simulator. By default, a log of the simulation is written to the *transcript* file in the current directory.

Incremental compilation

By default, ModelSim Verilog supports incremental compilation of designs, thus saving compilation time when you modify your design. Unlike other Verilog simulators, there is no requirement that you compile the entire design in one invocation of the compiler .

You are not required to compile your design in any particular order because all module and UDP instantiations and external hierarchical references are resolved when the design is loaded by the simulator. Incremental compilation is made possible by deferring these bindings, and as a result some errors cannot be detected during compilation. Commonly, these errors include: modules that were referenced but not compiled, incorrect port connections, and incorrect hierarchical references.

The following example shows how a hierarchical design can be compiled in top-down order:

Contents of top.v:

```
module top;
    or2 or2_i (n1, a, b);
    and2 and2_i (n2, n1, c);
endmodule
```

Contents of and2.v:

```
module and2(y, a, b);
    output y;
    input a, b;
    and(y, a, b);
endmodule
```

Contents of or2.v:

```
module or2(y, a, b);
    output y;
    input a, b;
    or(y, a, b);
endmodule
```

Compile the design in top down order (assumes work library already exists):

```
% vlog top.v
-- Compiling module top

Top level modules:
    top
% vlog and2.v
-- Compiling module and2

Top level modules:
    and2
% vlog or2.v
-- Compiling module or2

Top level modules:
    or2
```

Note that the compiler lists each module as a top level module, although, ultimately, only *top* is a top-level module. If a module is not referenced by another module compiled in the same invocation of the compiler, then it is listed as a top level module. This is just an

informative message and can be ignored during incremental compilation. The message is more useful when you compile an entire design in one invocation of the compiler and need to know the top-level module names for the simulator. For example,

```
% vlog top.v and2.v or2.v
-- Compiling module top
-- Compiling module and2
-- Compiling module or2

Top level modules:
    top
```

The most efficient method of incremental compilation is to manually compile only the modules that have changed. This is not always convenient, especially if your source files have compiler directive interdependencies (such as macros). In this case, you may prefer to always compile your entire design in one invocation of the compiler. If you specify the **-incr** argument, the compiler will automatically determine which modules have changed and generate code only for those modules. This is not as efficient as manual incremental compilation because the compiler must scan all of the source code to determine which modules must be compiled.

The following is an example of how to compile a design with automatic incremental compilation:

```
% vlog -incr top.v and2.v or2.v
-- Compiling module top
-- Compiling module and2
-- Compiling module or2

Top level modules:
    top
```

Now, suppose that you modify the functionality of the *or2* module:

```
% vlog -incr top.v and2.v or2.v
-- Skipping module top
-- Skipping module and2
-- Compiling module or2

Top level modules:
    top
```

The compiler informs you that it skipped the modules *top* and *and2*, and compiled *or2*.

Automatic incremental compilation is intelligent about when to compile a module. For example, changing a comment in your source code does not result in a recompile; however, changing the compiler command line arguments results in a recompile of all modules.

Note: Changes to your source code that do not change functionality but that do affect source code line numbers (such as adding a comment line) *will* cause all affected modules to be recompiled. This happens because debug information must be kept current so that ModelSim can trace back to the correct areas of the source code.

Library usage

All modules and UDPs in a Verilog design must be compiled into one or more libraries. One library is usually sufficient for a simple design, but you may want to organize your modules into various libraries for a complex design. If your design uses different modules having the same name, then you are required to put those modules in different libraries because design unit names must be unique within a library.

The following is an example of how you may organize your ASIC cells into one library and the rest of your design into another:

```
% vlib work
% vlib asiclib
% vlog -work asiclib and2.v or2.v
-- Compiling module and2
-- Compiling module or2

Top level modules:
    and2
    or2
% vlog top.v
-- Compiling module top

Top level modules:
    top
```

Note that the first compilation uses the **-work asiclib** argument to instruct the compiler to place the results in the **asiclib** library rather than the default **work** library.

Since instantiation bindings are not determined at compile time, you must instruct the simulator to search your libraries when loading the design. The top-level modules are loaded from the library named **work** unless you prefix the modules with the **<library>.** option. All other Verilog instantiations are resolved in the following order:

- Search libraries specified with **-Lf** arguments in the order they appear on the command line.
- Search the library specified in the "Verilog-XL `uselib compiler directive" (UM-74).
- Search libraries specified with **-L** arguments in the order they appear on the command line.
- Search the **work** library.
- Search the library explicitly named in the special escaped identifier instance name.

The work library is not necessarily a library named **work**—rather, the **work** library refers to the library containing the module that instantiates the module or UDP that is currently being searched for. This definition is useful if you have hierarchical modules organized into separate libraries and if sub-module names overlap among the libraries. In this situation you want the modules to search for their sub-modules in the work library first. This is accomplished by specifying **-L work** first in the list of search libraries.

For example, assume you have a top-level module *top* that instantiates module *modA* from library *libA* and module *modB* from library *libB*. Furthermore, *modA* and *modB* both instantiate modules named *cellA*, but the definition of *cellA* compiled into *libA* is different from that compiled into *libB*. In this case, it is insufficient to just specify **-L libA - L libB** as the search libraries because instantiations of *cellA* from *modB* resolve to the *libA* version of *cellA*. The appropriate search library arguments are **-L work -L libA -L libB**.

Verilog-XL compatible compiler arguments

The compiler arguments listed below are equivalent to Verilog-XL arguments and may ease the porting of a design to ModelSim. See the **vlog** command (CR-181) for a description of each argument.

```
+define+<macro_name>[=<macro_text>]
+delay_mode_distributed
+delay_mode_path
+delay_mode_unit
+delay_mode_zero
-f <filename>
+incdir+<directory>
+mindelays
+maxdelays
+nowarn<mnemonic>
+typdelays
-u
```

Arguments supporting source libraries

The compiler arguments listed below support source libraries in the same manner as Verilog-XL. See the **vlog** command (CR-181) for a description of each argument.

Note that these source libraries are very different from the libraries that the ModelSim compiler uses to store compilation results. You may find it convenient to use these arguments if you are porting a design to ModelSim or if you are familiar with these arguments and prefer to use them.

Source libraries are searched after the source files on the command line are compiled. If there are any unresolved references to modules or UDPs, then the compiler searches the source libraries to satisfy them. The modules compiled from source libraries may in turn have additional unresolved references that cause the source libraries to be searched again. This process is repeated until all references are resolved or until no new unresolved references are found. Source libraries are searched in the order they appear on the command line.

```
-v <filename>
-y <directory>
+libext+<suffix>
+librescan
+nolibcell
-R [<simargs>]
```

Verilog-XL ‘uselib compiler directive

The **‘uselib** compiler directive is an alternative source library management scheme to the **-v**, **-y**, and **+libext** compiler arguments. It has the advantage that a design may reference different modules having the same name. You compile designs that contain **‘uselib** directive statements using the **-compile_uselibs** argument (described below) to **vlog** (CR-181).

The syntax for the **‘uselib** directive is:

```
‘uselib <library_reference>...
```

where <library_reference> is:

```
dir=<library_directory> | file=<library_file> | libext=<file_extension> |
lib=<library_name>
```

The library references are equivalent to command line arguments as follows:

```
dir=<library_directory> -y <library_directory>
file=<library_file> -v <library_file>
libext=<file_extension> +libext+<file_extension>
```

For example, the following directive

```
‘uselib dir=/h/vendorA libext=.v
```

is equivalent to the following command line arguments:

```
-y /h/vendorA +libext+.v
```

Since the **‘uselib** directives are embedded in the Verilog source code, there is more flexibility in defining the source libraries for the instantiations in the design. The appearance of a **‘uselib** directive in the source code explicitly defines how instantiations that follow it are resolved, completely overriding any previous **‘uselib** directives.

-compile_uselibs argument

Use the **-compile_uselibs** argument to **vlog** (CR-181) to reference **‘uselib** directives. The argument finds the source files referenced in the directive, compiles them into automatically created object libraries, and updates the *modelsim.ini* file with the logical mappings to the libraries.

When using **-compile_uselibs**, ModelSim determines into what directory to compile the object libraries by choosing, in order, from the following three values:

- The directory name specified by the **-compile_uselibs** argument. For example, `-compile_uselibs=./mydir`
- The directory specified by the MTI_USELIB_DIR environment variable (see "Environment variables" (UM-337))
- A directory named *mti_uselibs* that is created in the current working directory

Note: In ModelSim versions prior to 5.5, the library files referenced by the **‘uselib** directive were not automatically compiled by ModelSim Verilog. To maintain backwards compatibility, this is still the default behavior when **-compile_uselibs** is not used. See www.model.com/products/documentation/pre55_uselib.pdf for a description of the pre-5.5 implementation.

The following code fragment and compiler invocation show how two different modules that have the same name can be instantiated within the same design:

```
module top;
  ‘uselib dir=/h/vendorA libext=.v
  NAND2 u1(n1, n2, n3);
  ‘uselib dir=/h/vendorB libext=.v
  NAND2 u2(n4, n5, n6);
endmodule
```

This allows the NAND2 module to have different definitions in the vendorA and vendorB libraries.

‘uselib is persistent

As mentioned above, the appearance of a **‘uselib** directive in the source code explicitly defines how instantiations that follow it are resolved. This may result in unexpected consequences. For example, consider the following compile command:

```
vlog -compile_uselibs dut.v srtr.v
```

Assume that *dut.v* contains a **‘uselib** directive. Since *srtr.v* is compiled after *dut.v*, the **‘uselib** directive is still in effect. When *srtr* is loaded it is using the **‘uselib** directive from *dut.v* to decide where to locate modules. If this is not what you intend, then you need to put an empty **‘uselib** at the end of *dut.v* to "close" the previous **‘uselib** statement.

Simulation

The ModelSim simulator can load and simulate both Verilog and VHDL designs, providing a uniform graphic interface and simulation control commands for debugging and analyzing your designs. The graphic interface and simulator commands are described elsewhere in this manual, while this section focuses specifically on Verilog simulation.

Invoking the simulator

A Verilog design is ready for simulation after it has been compiled into one or more libraries. The simulator may then be invoked with the names of the top-level modules (many designs contain only one top level module). For example, if your top level modules are "testbench" and "globals", then invoke the simulator as follows:

```
vsim testbench globals
```

After the simulator loads the top-level modules, it iteratively loads the instantiated modules and UDPs in the design hierarchy, linking the design together by connecting the ports and resolving hierarchical references. By default all modules and UDPs are loaded from the library named **work**. Modules and UDPs from other libraries can be specified using the **-L** or **-Lf** arguments to **vsim** (see "Library usage" (UM-72) for details).

On successful loading of the design, the simulation time is set to zero, and you must enter a **run** command to begin simulation. Commonly, you enter **run -all** to run until there are no more simulation events or until **$finish** is executed in the Verilog code. You can also run for specific time periods (e.g., `run 100 ns`). Enter the **quit** command to exit the simulator.

Simulator resolution limit

The simulator internally represents time as a 64-bit integer in units equivalent to the smallest unit of simulation time, also known as the simulator resolution limit. The resolution limit defaults to the smallest time precision found among all of the **‘timescale** compiler directives in the design. Here is an example of a **‘timescale** directive:

```
‘timescale 1 ns / 100 ps
```

The first number is the time units and the second number is the time precision. The directive above causes time values to be read as ns and to be rounded to the nearest 100 ps.

Modules without timescale directives

You may encounter unexpected behavior if your design contains some modules with timescale directives and others without. The time units for modules without a timescale directive default to the simulator resolution. For example, say you have the two modules shown in the table below:

Module 1	**Module 2**

`timescale 1 ns / 10 ps
 	
module mod1 (set);
 	
output set;
reg set;
parameter d = 1.55;
 	
initial
begin
set = 1'bz;
#d set = 1'b0;
#d set = 1'b1;
end
 	
endmodule
	module mod2 (set);
	
	output set;
	reg set;
	parameter d = 1.55;
	
	initial
	begin
	set = 1'bz;
	#d set = 1'b0;
	#d set = 1'b1;
	end
	
	endmodule

If you invoke **vsim** as `vsim mod2 mod1` then Module 1 sets the simulator resolution to 10 ps. Module 2 has no timescale directive, so the time units default to the simulator resolution, in this case 10 ps. If you watched */mod1/set* and */mod2/set* in the Wave window, you’d see that in Module 1 it transitions every 1.55 ns as expected (because of the 1 ns time unit in the timescale directive). However, in Module 2, *set* transitions every 20 ps. That’s because the delay of 1.55 in Module 2 is read as 15.5 ps and is rounded up to 20 ps.

In such cases ModelSim will issue the following warning message during elaboration:

```
** Warning: (vsim-3010) [TSCALE] - Module ’mod1’ has a ‘timescale directive
in effect, but previous modules do not.
```

If you invoke **vsim** as `vsim mod1 mod2`, the simulation results would be the same but ModelSim would produce a different warning message:

```
** Warning: (vsim-3009) [TSCALE] - Module 'mod2' does not have a 'timescale
directive in effect, but previous modules do.
```

These warnings should ALWAYS be investigated.

If the design contains no 'timescale directives, then the resolution limit and time units default to the value specified by the **Resolution** (UM-347) variable in the modelsim.ini file. (The variable is set to 1 ps by default.)

Multiple timescale directives

As alluded to above, your design can have multiple timescale directives. The timescale directive takes effect where it appears in a source file and applies to all source files which follow in the same **vlog** (CR-181) command. Separately compiled modules can also have different timescales. The simulator determines the smallest timescale of all the modules in a design and uses that as the simulator resolution.

Overriding the resolution

You can override the simulator resolution (or ModelSim's default resolution) by specifying the **-t** argument on the command line or by selecting a different Simulator Resolution in the **Simulate** dialog box. Available resolutions are: 1x, 10x or 100x of fs, ps, ns, us, ms, or sec.

For example this command chooses 10 ps resolution:

```
vsim -t 10ps top
```

Clearly you need to be careful when doing this type of operation. If the resolution set by **-t** is larger than the timescale of some module, the time values in that module are rounded to the next multiple of the resolution. In the example above, a delay of 4 ps would be rounded to 0 ps.

Choosing the resolution

You should choose the coarsest resolution limit possible that does not result in undesired rounding of your delays. The time precision should not be unnecessarily small because it will limit the maximum simulation time limit, and it will degrade performance in some cases.

Event ordering in Verilog designs

Event-based simulators such as ModelSim may process multiple events at a given simulation time. The Verilog language is defined such that you cannot explicitly control the order in which simultaneous events are processed. Unfortunately, some designs rely on a particular event order, and these designs may behave differently than you expect.

Event queues

Section 5 of the IEEE Std 1364-1995 LRM defines several event queues that determine the order in which events are evaluated. At the current simulation time, the simulator has the following pending events:

- active events
- inactive events
- non-blocking assignment update events
- monitor events
- future events
  - inactive events
  - non-blocking assignment update events

The LRM dictates that events are processed as follows – 1) all active events are processed; 2) the inactive events are moved to the active event queue and then processed; 3) the non-blocking events are moved to the active event queue and then processed; 4) the monitor events are moved to the active queue and then processed; 5) simulation advances to the next time where there is an inactive event or a non-blocking assignment update event.

Within the active event queue, the events can be processed in any order, and new active events can be added to the queue in any order. In other words, you *cannot* control event order within the active queue. The example below illustrates potential ramifications of this situation.

Say you have these four statements:

1 always@(q) p = q;

2 always @(q) p2 = not q;

3 always @(p or p2) clk = p and p2;

4 always @(posedge clk)

and current values as follows: q = 0, p = 0, p2=1

The tables below show two of the many valid evaluations of these statements. Evaluation events are denoted #, where # is the statement to be evaluated. Update events are denoted *<name>(old->new)* where *<name>* indicates the reg being updated and *new* is the updated value.

Table 1: Evaluation 1

Event being processed	Active event queue
	q(0 → 1)
q(0 → 1)	1, 2
1	p(0 → 1), 2
p(0 → 1)	3, 2
3	clk(0 → 1), 2
clk(0 → 1)	4, 2
4	2
2	p2(1 → 0)
p2(1 → 0)	3
3	clk(1 → 0)
clk(1 → 0)	<empty>

Table 2: Evaluation 2

Event being processed	Active event queue
	q(0 → 1)
q(0 → 1)	1, 2
1	p(0 → 1), 2
2	p2(1 → 0), p(0 → 1)
p(0 → 1)	3, p2(1 → 0)
p2(1 → 0)	3
3	<empty> (clk doesn't change)

Again, both evaluations are valid. However, in Evaluation 1, *clk* has a glitch on it; in Evaluation 2, *clk* doesn't. This indicates that the design has a zero-delay race condition on *clk*.

'Controlling' event queues with blocking/non-blocking assignments

The only control you have over event order is to assign an event to a particular queue. You do this via blocking or non-blocking assignments.

Blocking assignments

Blocking assignments place an event in the active, inactive, or future queues depending on what type of delay they have:

- a blocking assignment without a delay goes in the active queue
- a blocking assignment with an explicit delay of 0 goes in the inactive queue
- a blocking assignment with a non-zero delay goes in the future queue

Non-blocking assignments

A non-blocking assignment goes into either the non-blocking assignment update event queue or the future non-blocking assignment update event queue. (Non-blocking assignments with no delays and those with explicit zero delays are treated the same.)

Non-blocking assignments should be used only for outputs of flip-flops. This insures that all outputs of flip-flops do not change until after all flip-flops have been evaluated. Attempting to use non-blocking assignments in combinational logic paths to remove race conditions may only cause more problems. (In the preceding example, changing all statements to non-blocking assignments would not remove the race condition.) This includes using non-blocking assignments in the generation of gated clocks.

The following is an example of how to properly use non-blocking assignments.

```
gen1: always @(master)
  clk1 = master;

gen2: always @(clk1)
  clk2 = clk1;

f1 : always @(posedge clk1)
  begin
    q1 <= d1;
  end

f2:   always @(posedge clk2)
  begin
    q2 <= q1;
  end
```

If written this way, a value on *d1* always takes two clock cycles to get from *d1* to *q2*. If you change *clk1 = master* and *clk2 = clk1* to non-blocking assignments or *q2 <= q1* and *q1 <= d1* to blocking assignments, then *d1* may get to *q2* is less than two clock cycles.

Debugging event order issues

Since many models have been developed on Verilog-XL, ModelSim tries to duplicate Verilog-XL event ordering to ease the porting of those models to ModelSim. However, ModelSim does not match Verilog-XL event ordering in all cases, and if a model ported to ModelSim does not behave as expected, then you should suspect that there are event order dependencies.

ModelSim helps you track down event order dependencies with the following compiler arguments: **-compat**, **-hazards**, and **-keep_delta**.

See the **vlog** command (CR-181) for descriptions of **-compat** and **-keep_delta**.

Hazard detection

The **-hazard** argument to **vsim** (CR-189) detects event order hazards involving simultaneous reading and writing of the same register in concurrently executing processes. **vsim** detects the following kinds of hazards:

- WRITE/WRITE:
  Two processes writing to the same variable at the same time.
- READ/WRITE:
  One process reading a variable at the same time it is being written to by another process. ModelSim calls this a READ/WRITE hazard if it executed the read first.
- WRITE/READ:
  Same as a READ/WRITE hazard except that ModelSim executed the write first.

vsim issues an error message when it detects a hazard. The message pinpoints the variable and the two processes involved. You can have the simulator break on the statement where the hazard is detected by setting the **break on assertion** level to **error**.

To enable hazard detection you must invoke **vlog** (CR-181) with the **-hazards** argument when you compile your source code and you must also invoke **vsim** with the **-hazards** argument when you simulate.

Important: Enabling **-hazards** implicitly enables the **-compat** argument. As a result, using this argument may affect your simulation results.

Limitations of hazard detection

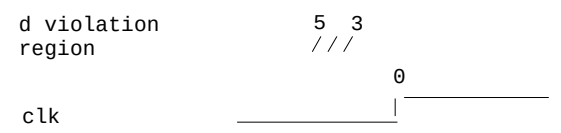
- Reads and writes involving bit and part selects of vectors are not considered for hazard detection. The overhead of tracking the overlap between the bit and part selects is too high.
- A WRITE/WRITE hazard is flagged even if the same value is written by both processes.
- A WRITE/READ or READ/WRITE hazard is flagged even if the write does not modify the variable's value.
- Glitches on nets caused by non-guaranteed event ordering are not detected.

Negative timing check limits

Verilog supports negative limit values in the $setuphold and $recrem system tasks. These tasks have optional delayed versions of input signals to insure proper evaluation of models with negative timing check limits. Delay values for these delayed nets are determined by the simulator so that valid data is available for evaluation before a clocking signal.

Example

```
$setuphold(posedge clk, negedge d, 5, -3, Notifier,,, clk_dly, d_dly);;
```



ModelSim calculates the delay for signal *d_dly* as 4 time units instead of 3. It does this to prevent *d_dly* and *clk_dly* from occurring simultaneously when a violation isn't reported.

Note: ModelSim accepts negative limit checks by default, unlike current versions of Verilog-XL. To match Verilog-XL default behavior (i.e., zeroing all negative timing check limits), use the **+no_neg_tcheck** argument to **vsim** (CR-189).

Negative timing constraint algorithm

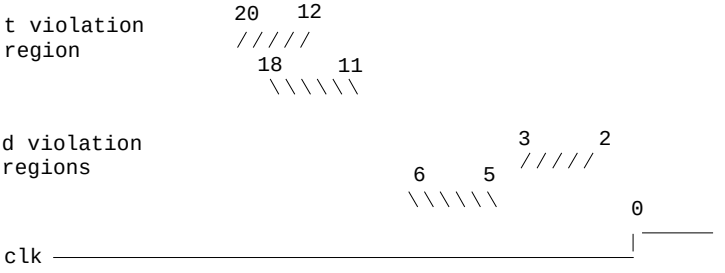
The algorithm ModelSim uses to calculate delays for delayed nets isn't described in IEEE Std 1364. Rather, ModelSim matches Verilog-XL behavior. The algorithm attempts to find a set of delays so the data net is valid when the clock net transitions and the timing checks are satisfied. The algorithm is iterative because a set of delays can be selected that satisfies all timing checks for a pair of inputs but then causes mis-ordering of another pair (where both pairs of inputs share a common input). When a set of delays that satisfies all timing checks is found, the delays are said to converge.

When none of the delay sets cause convergence, the algorithm pessimistically changes the timing check limits to force convergence. Basically the algorithm zeroes the smallest negative $setup/$recovery limit. If a negative $setup/$recovery doesn't exist, then the algorithm zeros the smallest negative $hold/$removal limit. After zeroing a negative limit, the delay calculation procedure is repeated. If the delays don't converge, the algorithm zeros another negative limit, repeating the process until convergence is found.

A simple example will help clarify the algorithm. Assume you have the following timing checks:

```
$setuphold(posedge clk, posedge d, 3, -2 , NOTIFIER,,, clk_dly, d_dly);
$setuphold(posedge clk, negedge d, 6, -5 , NOTIFIER,,, clk_dly, d_dly);
$setuphold(posedge clk, posedge t, 20, -12 , NOTIFIER,,, clk_dly, t_dly);
$setuphold(posedge clk, negedge t, 18, -11 , NOTIFIER,,, clk_dly, t_dly);
```

The violation regions for t and d in this example are:



Note that the delays between *clk/clk_dly*, *t/t_dly*, and *d/d_dly* are not edge sensitive, and they must be the same for both rising and falling transitions of *clk*, *t*, and *d*. A *d* => *d_dly* delay of 5 will satisfy the negedge case (transitions of *d* from 5 to 0 before *clk* won't be latched), but valid transitions of posedge *d*, in the region of 5 to 3 before *clk*, won't latch correctly. Therefore, to find convergence, the algorithm starts zeroing negative **$hold** limits (-12, then -11, and then -5). The check limits on *t* are zeroed first because of their magnitude.

ModelSim will display messages when limits are zeroed if you use the **+ntc_warn** argument. Even if you don't set **+ntc_warn**, ModelSim displays a summary of any zeroed limits.

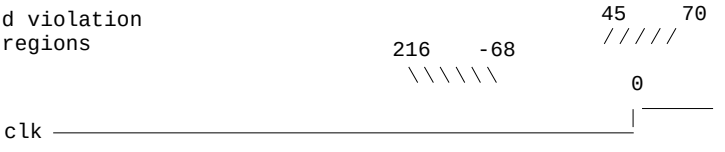
Extending check limits without zeroing

If zeroing limits is too pessimistic for your design, you can use the **vsim** (CR-189) arguments **-extend_tcheck_data_limit** and **-extend_tcheck_ref_limit** instead. These arguments cause a one-time extension of qualifying data or reference limits in an attempt to provide a solution prior to any limit zeroing. A limit qualifies if it bounds a violation region which does not overlap a related violation region.

An example will help illustrate. Assume you have the following timing checks:

```
$setuphold( posedge clk, posedge d,   45,  70, notifier,,,dclk,dd);
$setuphold( posedge clk, negedge d,  216, -68, notifier,,,dclk,dd);
```

The violation regions for d in this example are:



The delay net delay analysis in this case does not provide a solution. The required negative hold delay of 68 between *d* and *dd* could cause a non-violating posedge *d* transition to be delayed on *dd* so that it could arrive after *dclk* for functional evaluation. By default the -68 hold limit is set pessimistically to 0 to insure the correct functional evaluation.

Alternatively, you could use **-extend_tcheck_data_limit** to overlap the regions. In this example we must specify the percentage by which to "decrease" the negative hold limit in order to overlap the positive setup limit. In other words, you must extend the 216, -68 region to 216, -44. You would calculate the percentage as follows:

1 Calculate the size of the negative edge violation region:

216 - 68 = 148

2 Calculate the gap between the negative hold limit and the positive setup limit and add one timing unit to allow for overlap:

68 - 45 = 23 + 1 = 24

3 Divide the gap size by the violation region size:

24 / 148 = .16

Hence, you would set **-extend_tcheck_data_limit** to 16.

Note: ModelSim will extend the limit only as far as is needed to derive a solution. So if you used 100 in the previous example, it would still only extend the limit 16 percent. Indeed, in some cases it may be easiest to select a large percentage number and not worry about an exact calculation.

Verilog-XL compatible simulator arguments

The simulator arguments listed below are equivalent to Verilog-XL arguments and may ease the porting of a design to ModelSim. See the **vsim** (CR-189) for a description of each argument.

```
+alt_path_delays
-l <filename>
+maxdelays
+mindelays
+multisource_int_delays
+no_cancelled_e_msg
+no_neg_tchk
+no_notifier
+no_path_edge
+no_pulse_msg
+no_show_cancelled_e
+nosdfwarn
+nowarn<mnemonic>
+ntc_warn
+pulse_e/<percent>
+pulse_e_style_ondetect
+pulse_e_style_onevent
+pulse_int_e/<percent>
+pulse_int_r/<percent>
+pulse_r/<percent>
+sdf_nocheck_celltype
+sdf_verbose
+show_cancelled_e
+transport_int_delays
+transport_path_delays
+typdelays
```

Cell libraries

Model Technology passed the ASIC Council's Verilog test suite and achieved the "Library Tested and Approved" designation from Si2 Labs. This test suite is designed to ensure Verilog timing accuracy and functionality and is the first significant hurdle to complete on the way to achieving full ASIC vendor support. As a consequence, many ASIC and FPGA vendors' Verilog cell libraries are compatible with ModelSim Verilog.

The cell models generally contain Verilog "specify blocks" that describe the path delays and timing constraints for the cells. See section 13 in the IEEE Std 1364-1995 for details on specify blocks, and section 14.5 for details on timing constraints. ModelSim Verilog fully implements specify blocks and timing constraints as defined in IEEE Std 1364 along with some Verilog-XL compatible extensions.

SDF timing annotation

ModelSim Verilog supports timing annotation from Standard Delay Format (SDF) files. See *Chapter 9 - Standard Delay Format (SDF) Timing Annotation* for details.

Delay modes

Verilog models may contain both distributed delays and path delays. The delays on primitives, UDPs, and continuous assignments are the distributed delays, whereas the port-to-port delays specified in specify blocks are the path delays. These delays interact to determine the actual delay observed. Most Verilog cells use path delays exclusively, with the distributed delays set to zero. For example,

```
module and2(y, a, b);
    input a, b;
    output y;

    and(y, a, b);

    specify
        (a => y) = 5;
        (b => y) = 5;
    endspecify
endmodule
```

In the above two-input "and" gate cell, the distributed delay for the "and" primitive is zero, and the actual delays observed on the module ports are taken from the path delays. This is typical for most cells, but a complex cell may require non-zero distributed delays to work properly. Even so, these delays are usually small enough that the path delays take priority over the distributed delays. The rule is that if a module contains both path delays and distributed delays, then the larger of the two delays for each path shall be used (as defined by the IEEE Std 1364). This is the default behavior, but you can specify alternate delay modes with compiler directives and arguments. These arguments and directives are compatible with Verilog-XL. Compiler delay mode arguments take precedence over delay mode directives in the source code.

Distributed delay mode

In distributed delay mode the specify path delays are ignored in favor of the distributed delays. Select this delay mode with the **+delay_mode_distributed** compiler argument or the **‘delay_mode_distributed** compiler directive.

Path delay mode

In path delay mode the distributed delays are set to zero in any module that contains a path delay. Select this delay mode with the **+delay_mode_path** compiler argument or the **‘delay_mode_path** compiler directive.

Unit delay mode

In unit delay mode the distributed delays are set to one (the unit is the time_unit specified in the **‘timescale** directive), and the specify path delays and timing constraints are ignored. Select this delay mode with the **+delay_mode_unit** compiler argument or the **‘delay_mode_unit** compiler directive.

Zero delay mode

In zero delay mode the distributed delays are set to zero, and the specify path delays and timing constraints are ignored. Select this delay mode with the **+delay_mode_zero** compiler argument or the **‘delay_mode_zero** compiler directive.

System tasks

The IEEE Std 1364 defines many system tasks as part of the Verilog language, and ModelSim Verilog supports all of these along with several non-standard Verilog-XL system tasks. The system tasks listed in this chapter are built into the simulator, although some designs depend on user-defined system tasks implemented with the Programming Language Interface (PLI) or Verilog Procedural Interface (VPI). If the simulator issues warnings regarding undefined system tasks, then it is likely that these system tasks are defined by a PLI/VPI application that must be loaded by the simulator.

IEEE Std 1364 system tasks

The following system tasks are described in detail in the IEEE Std 1364.

Timescale tasks	Simulator control tasks	Simulation time functions	Command line input
$printtimescale	$finish	$realtime	$test$plusargs
$timeformat	$stop	$stime	$value$plusargs
		$time	

Probabilistic distribution functions	Conversion functions	Stochastic analysis tasks	Timing check tasks
$dist_chi_square	$bitstoreal	$q_add	$hold
$dist_erlang	$itor	$q_exam	$nochange
$dist_exponential	$realtobits	$q_full	$period
$dist_normal	$rtoi	$q_initialize	$recovery
$dist_poisson	$signed	$q_remove	$setup
$dist_t	$unsigned		$setuphold
$dist_uniform			$skew
$random			$width
			$removal
			$recrem

Display tasks	PLA modeling tasks	Value change dump (VCD) file tasks
$display	$async$and$array	$dumpall
$displayb	$async$nand$array	$dumpfile
$displayh	$async$or$array	$dumpflush
$displayo	$async$nor$array	$dumplimit
$monitor	$async$and$plane	$dumpoff
$monitorb	$async$nand$plane	$dumpon
$monitorh	$async$or$plane	$dumpvars
$monitoro	$async$nor$plane	
$monitoroff	$sync$and$array	
$monitoron	$sync$nand$array	
$strobe	$sync$or$array	
$strobeb	$sync$nor$array	
$strobeh	$sync$and$plane	
$strobeo	$sync$nand$plane	
$write	$sync$or$plane	
$writeb	$sync$nor$plane	
$writeh		
$writeo		

File I/O tasks

$fclose	$fopen	$fwriteh
$fdisplay	$fread	$fwriteo
$fdisplayb	$fscanf	$readmemb
$fdisplayh	$fseek	$readmemh
$fdisplayo	$fstrobe	$rewind
$ferror	$fstrobeb	$sdf_annotate
$fflush	$fstrobeh	$sformat
$fgetc	$fstrobeo	$sscanf
$fgets	$ftell	$swrite
$fmonitor	$fwrite	$swriteb
$fmonitorb	$fwriteb	$swriteh
$fmonitorh		$swriteo
$fmonitoro		$ungetc

Note: $readmemb and $readmemh match the behavior of Verilog-XL rather than IEEE Std 1364. Specifically, they load data into memory starting with the lowest address. For example, whether you make the declaration `memory[127:0]` or `memory[0:127]`, ModelSim will load data starting at address 0 and work upwards to address 127.

Verilog-XL compatible system tasks

The following system tasks are provided for compatibility with Verilog-XL. Although they are not part of the IEEE standard, they are described in an annex of the IEEE Std 1364.

```
$countdrivers
$getpattern
$sreadmemb
$sreadmemh
```

The following system tasks are also provided for compatibility with Verilog-XL; they are not described in the IEEE Std 1364.

`$deposit(variable, value);`
This system task sets a Verilog register or net to the specified value. **variable** is the register or net to be changed; **value** is the new value for the register or net. The value remains until there is a subsequent driver transaction or another $deposit task for the same register or net. This system task operates identically to the ModelSim **force -deposit** command.

`$disable_warnings("<keyword>"?<,<module_instance>>*?);`
This system task instructs ModelSim to disable warnings about timing check violations or triregs that acquire a value of 'X' due to charge decay. <keyword> may be decay or timing. If you don't specify a module_instance, ModelSim disables warnings for the entire simulation.

`$enable_warnings("<keyword>"?<,<module_instance>>*?);`
This system task enables warnings about timing check violations or triregs that acquire a value of 'X' due to charge decay. <keyword> may be decay or timing. If you don't specify a module_instance, ModelSim enables warnings for the entire simulation.

The following system tasks are extended to provide additional functionality for negative timing constraints and an alternate method of conditioning, as in Verilog-XL.

```
$recovery(reference event, data_event, removal_limit, recovery_limit,
[notifier], [tstamp_cond], [tcheck_cond], [delayed_reference],
[delayed_data])
```
The $recovery system task normally takes a recovery_limit as the third argument and an optional notifier as the fourth argument. By specifying a limit for both the third and fourth arguments, the $recovery timing check is transformed into a combination removal and recovery timing check similar to the $recrem timing check. The only difference is that the removal_limit and recovery_limit are swapped.

```
$setuphold(clk_event, data_event, setup_limit, hold_limit, [notifier],
[tstamp_cond], [tcheck_cond], [delayed_clk], [delayed_data])
```
The tstamp_cond argument conditions the data_event for the setup check and the clk_event for the hold check. This alternate method of conditioning precludes specifying conditions in the clk_event and data_event arguments.

The tcheck_cond argument conditions the data_event for the hold check and the clk_event for the setup check. This alternate method of conditioning precludes specifying conditions in the clk_event and data_event arguments.

The delayed_clk argument is a net that is continuously assigned the value of the net specified in the clk_event. The delay is non-zero if the setup_limit is negative, zero otherwise.

The delayed_data argument is a net that is continuously assigned the value of the net specified in the data_event. The delay is non-zero if the hold_limit is negative, zero otherwise.

The delayed_clk and delayed_data arguments are provided to ease the modeling of devices that may have negative timing constraints. The model's logic should reference the delayed_clk and delayed_data nets in place of the normal clk and data nets. This ensures that the correct data is latched in the presence of negative constraints. The simulator automatically calculates the delays for delayed_clk and delayed_data such that the correct data is latched as long as a timing constraint has not been violated. See "Negative timing check limits" (UM-83) for more details.

The following system tasks are Verilog-XL system tasks that are not implemented in ModelSim Verilog, but have equivalent simulator commands.

`$input("filename")`
This system task reads commands from the specified filename. The equivalent simulator command is **do <filename>**.

`$list[(hierarchical_name)]`
This system task lists the source code for the specified scope. The equivalent functionality is provided by selecting a module in the graphic interface Structure window. The corresponding source code is displayed in the Source window.

`$reset`
This system task resets the simulation back to its time 0 state. The equivalent simulator command is **restart**.

`$restart("filename")`
This system task sets the simulation to the state specified by filename, saved in a previous call to $save. The equivalent simulator command is **restore <filename>**.

`$save("filename")`
This system task saves the current simulation state to the file specified by filename. The equivalent simulator command is **checkpoint <filename>**.

`$scope(hierarchical_name)`
This system task sets the interactive scope to the scope specified by hierarchical_name. The equivalent simulator command is **environment <pathname>**.

`$showscopes`
This system task displays a list of scopes defined in the current interactive scope. The equivalent simulator command is **show**.

`$showvars`
This system task displays a list of registers and nets defined in the current interactive scope. The equivalent simulator command is **show**.

ModelSim Verilog system tasks

The following system tasks are specific to ModelSim. They are not included in the IEEE Std 1364 nor are they likely supported in other simulators. Their use may limit the portability of your code.

`$init_signal_driver`
The $init_signal_driver() system task drives the value of a VHDL signal or Verilog net onto an existing VHDL signal or Verilog net. This allows you to drive signals or nets at any level of the design hierarchy from within a Verilog module (e.g., a testbench). See $init_signal_driver (UM-280) in *Chapter 8 - Signal Spy* for complete details and syntax on this system task.

`$init_signal_spy`
The $init_signal_spy() system task mirrors the value of a VHDL signal or Verilog register/net onto an existing Verilog register or VHDL signal. This system task allows you to reference signals, registers, or nets at any level of hierarchy from within a Verilog module (e.g., a testbench). See $init_signal_spy (UM-283) in *Chapter 8 - Signal Spy* for complete details and syntax on this system task.

`$signal_force`
The $signal_force() system task forces the value specified onto an existing VHDL signal or Verilog register or net. This allows you to force signals, registers, or nets at any level of the design hierarchy from within a Verilog module (e.g., a testbench). A $signal_force works the same as the **force** command (CR-82) with the exception that you cannot issue a repeating force. See $signal_force (UM-285) in *Chapter 8 - Signal Spy* for complete details and syntax on this system task.

`$signal_release`
The $signal_release() system task releases a value that had previously been forced onto an existing VHDL signal or Verilog register or net. A $signal_release works the same as the **noforce** command (CR-92). See $signal_release (UM-287) in *Chapter 8 - Signal Spy* for complete details and syntax on this system task.

`$sdf_done`
This task is a "cleanup" function that removes internal buffers, called MIPDs, that have a delay value of zero. These MIPDs are inserted in response to the **-v2k_int_delay** argument to the **vsim** command (CR-189). In general the simulator will automatically remove all zero delay MIPDs. However, if you have $sdf_annotate() calls in your design that are not getting executed, the zero-delay MIPDs are not removed. Adding the $sdf_done task after your last $sdf_annotate() will remove any zero-delay MIPDs that have been created.

Compiler directives

ModelSim Verilog supports all of the compiler directives defined in the IEEE Std 1364, some Verilog-XL compiler directives, and some that are proprietary.

Many of the compiler directives (such as **‘timescale**) take effect at the point they are defined in the source code and stay in effect until the directive is redefined or until it is reset to its default by a **‘resetall** directive. The effect of compiler directives spans source files, so the order of source files on the compilation command line could be significant. For example, if you have a file that defines some common macros for the entire design, then you might need to place it first in the list of files to be compiled.

The **‘resetall** directive affects only the following directives by resetting them back to their default settings (this information is not provided in the IEEE Std 1364):

```
‘celldefine
‘default_decay_time
`default_nettype
`delay_mode_distributed
`delay_mode_path
`delay_mode_unit
`delay_mode_zero
`protected
`timescale
`unconnected_drive
`uselib
```

ModelSim Verilog implicitly defines the following macro:

```
`define MODEL_TECH
```

IEEE Std 1364 compiler directives

The following compiler directives are described in detail in the IEEE Std 1364.

```
`celldefine
`default_nettype
`define
`else
`elsif
`endcelldefine
`endif
`ifdef
‘ifndef
`include
‘line
`nounconnected_drive
`resetall
`timescale
`unconnected_drive
`undef
```

Verilog-XL compatible compiler directives

The following compiler directives are provided for compatibility with Verilog-XL.

`‘default_decay_time <time>`
This directive specifies the default decay time to be used in trireg net declarations that do not explicitly declare a decay time. The decay time can be expressed as a real or integer number, or as "infinite" to specify that the charge never decays.

`` `delay_mode_distributed ``
This directive disables path delays in favor of distributed delays. See "Delay modes" (UM-87) for details.

`` `delay_mode_path ``
This directive sets distributed delays to zero in favor of path delays. See "Delay modes" (UM-87) for details.

`` `delay_mode_unit ``
This directive sets path delays to zero and non-zero distributed delays to one time unit. See Delay modes (UM-87) for details.

`` `delay_mode_zero ``
This directive sets path delays and distributed delays to zero. See "Delay modes" (UM-87) for details.

`` `uselib ``
This directive is an alternative to the **-v**, **-y**, and **+libext** source library compiler arguments. See "Verilog-XL ‘uselib compiler directive" (UM-74) for details.

The following Verilog-XL compiler directives are silently ignored by ModelSim Verilog. Many of these directives are irrelevant to ModelSim Verilog, but may appear in code being ported from Verilog-XL.

```
`accelerate
`autoexpand_vectornets
`disable_portfaults
`enable_portfaults
`expand_vectornets
`noaccelerate
`noexpand_vectornets
`noremove_gatenames
`noremove_netnames
`nosuppress_faults
`remove_gatenames
`remove_netnames
`suppress_faults
```

The following Verilog-XL compiler directives produce warning messages in ModelSim Verilog. These are not implemented in ModelSim Verilog, and any code containing these directives may behave differently in ModelSim Verilog than in Verilog-XL.

```
`default_trireg_strength
`signed
`unsigned
```

Verilog PLI/VPI

The Verilog PLI (Programming Language Interface) and VPI (Verilog Procedural Interface) both provide a mechanism for defining system tasks and functions that communicate with the simulator through a C procedural interface. There are many third party applications available that interface to Verilog simulators through the PLI (see "Third party PLI applications" (UM-108)). In addition, you may write your own PLI/VPI applications.

ModelSim Verilog implements the PLI as defined in the IEEE Std 1364, with the exception of the **acc_handle_datapath()** routine. We did not implement the **acc_handle_datapath()** routine because the information it returns is more appropriate for a static timing analysis tool. The VPI is partially implemented as defined in the IEEE Std 1364-2001. The list of currently supported functionality can be found in the following file:

```
<install_dir>/modeltech/docs/technotes/Verilog_VPI.note
```

The IEEE Std 1364 is the reference that defines the usage of the PLI/VPI routines. This manual only describes details of using the PLI/VPI with ModelSim Verilog.

Registering PLI applications

Each PLI application must register its system tasks and functions with the simulator, providing the name of each system task and function and the associated callback routines. Since many PLI applications already interface to Verilog-XL, ModelSim Verilog PLI applications make use of the same mechanism to register information about each system task and function in an array of s_tfcell structures. This structure is declared in the veriuser.h include file as follows:

```
typedef int (*p_tffn)();

typedef struct t_tfcell {
    short type;/* USERTASK, USERFUNCTION, or USERREALFUNCTION */
    short data;/* passed as data argument of callback function */
    p_tffn checktf;  /* argument checking callback function */
    p_tffn sizetf;   /* function return size callback function */
    p_tffn calltf;   /* task or function call callback function */
    p_tffn misctf;   /* miscellaneous reason callback function */
    char *tfname;/* name of system task or function */

        /* The following fields are ignored by ModelSim Verilog */
    int forwref;
    char *tfveritool;
    char *tferrmessage;
    int hash;
    struct t_tfcell *left_p;
    struct t_tfcell *right_p;
    char *namecell_p;
    int warning_printed;
} s_tfcell, *p_tfcell;
```

The various callback functions (checktf, sizetf, calltf, and misctf) are described in detail in the IEEE Std 1364. The simulator calls these functions for various reasons. All callback functions are optional, but most applications contain at least the calltf function, which is called when the system task or function is executed in the Verilog code. The first argument to the callback functions is the value supplied in the data field (many PLI applications don't

use this field). The type field defines the entry as either a system task (USERTASK) or a system function that returns either a register (USERFUNCTION) or a real (USERREALFUNCTION). The tfname field is the system task or function name (it must begin with $). The remaining fields are not used by ModelSim Verilog.

On loading of a PLI application, the simulator first looks for an init_usertfs function, and then a veriusertfs array. If init_usertfs is found, the simulator calls that function so that it can call mti_RegisterUserTF() for each system task or function defined. The mti_RegisterUserTF() function is declared in veriuser.h as follows:

```
void mti_RegisterUserTF(p_tfcell usertf);
```

The storage for each usertf entry passed to the simulator must persist throughout the simulation because the simulator de-references the usertf pointer to call the callback functions. We recommend that you define your entries in an array, with the last entry set to 0. If the array is named veriusertfs (as is the case for linking to Verilog-XL), then you don't have to provide an init_usertfs function, and the simulator will automatically register the entries directly from the array (the last entry must be 0). For example,

```
s_tfcell veriusertfs[] = {
    {usertask, 0, 0, 0, abc_calltf, 0, "$abc"},
    {usertask, 0, 0, 0, xyz_calltf, 0, "$xyz"},
    {0}  /* last entry must be 0 */
};
```

Alternatively, you can add an init_usertfs function to explicitly register each entry from the array:

```
void init_usertfs()
{
    p_tfcell usertf = veriusertfs;
    while (usertf->type)
        mti_RegisterUserTF(usertf++);
}
```

It is an error if a PLI shared library does not contain a veriusertfs array or an init_usertfs function.

Since PLI applications are dynamically loaded by the simulator, you must specify which applications to load (each application must be a dynamically loadable library, see "Compiling and linking PLI/VPI C applications" (UM-101)). The PLI applications are specified as follows (note that on a Windows platform the file extension would be .dll):

- As a list in the Veriuser entry in the *modelsim.ini* file:

```
Veriuser = pliapp1.so pliapp2.so pliappn.so
```

- As a list in the PLIOBJS environment variable:

```
% setenv PLIOBJS "pliapp1.so pliapp2.so pliappn.so"
```

- As a -pli argument to the simulator (multiple arguments are allowed):

```
-pli pliapp1.so -pli pliapp2.so -pli pliappn.so
```

The various methods of specifying PLI applications can be used simultaneously. The libraries are loaded in the order listed above. Environment variable references can be used in the paths to the libraries in all cases.

Registering VPI applications

Each VPI application must register its system tasks and functions and its callbacks with the simulator. To accomplish this, one or more user-created registration routines must be called at simulation startup. Each registration routine should make one or more calls to vpi_register_systf() to register user-defined system tasks and functions and vpi_register_cb() to register callbacks. The registration routines must be placed in a table named vlog_startup_routines so that the simulator can find them. The table must be terminated with a 0 entry.

Example

```
PLI_INT32 MyFuncCalltf( PLI_BYTE8 *user_data )
{ ... }

PLI_INT32 MyFuncCompiletf( PLI_BYTE8 *user_data )
{ ... }

PLI_INT32 MyFuncSizetf( PLI_BYTE8 *user_data )
{ ... }

PLI_INT32 MyEndOfCompCB( p_cb_data cb_data_p )
{ ... }

PLI_INT32 MyStartOfSimCB( p_cb_data cb_data_p )
{ ... }

void RegisterMySystfs( void )

  {

      vpiHandle tmpH;
      s_cb_data callback;
      s_vpi_systf_data systf_data;

      systf_data.type        = vpiSysFunc;
      systf_data.sysfunctype = vpiSizedFunc;
      systf_data.tfname      = "$myfunc";
      systf_data.calltf      = MyFuncCalltf;
      systf_data.compiletf   = MyFuncCompiletf;
      systf_data.sizetf      = MyFuncSizetf;
      systf_data.user_data   = 0;
      tmpH = vpi_register_systf( &systf_data );
      vpi_free_object(tmpH);

      callback.reason    = cbEndOfCompile;
      callback.cb_rtn    = MyEndOfCompCB;
      callback.user_data = 0;
      tmpH = vpi_register_cb( &callback );
      vpi_free_object(tmpH);

      callback.reason    = cbStartOfSimulation;
      callback.cb_rtn    = MyStartOfSimCB;
      callback.user_data = 0;
      tmpH = vpi_register_cb( &callback );
      vpi_free_object(tmpH);
  }
void (*vlog_startup_routines[ ] ) () = {
    RegisterMySystfs,
      0    /* last entry must be 0 */
};
```

Loading VPI applications into the simulator is the same as described in "Registering PLI applications" (UM-97).

PLI and VPI applications can co-exist in the same application object file. In such cases, the applications are loaded at startup as follows:

- If an init_usertfs() function exists, then it is executed and only those system tasks and functions registered by calls to mti_RegisterUserTF() will be defined.
- If an init_usertfs() function does not exist but a veriusertfs table does exist, then only those system tasks and functions listed in the veriusertfs table will be defined.
- If an init_usertfs() function does not exist and a veriusertfs table does not exist, but a vlog_startup_routines table does exist, then only those system tasks and functions and callbacks registered by functions in the vlog_startup_routines table will be defined.

As a result, when PLI and VPI applications exist in the same application object file, they must be registered in the same manner. VPI registration functions that would normally be listed in a vlog_startup_routines table can be called from an init_usertfs() function instead.

Compiling and linking PLI/VPI C applications

The following platform-specific instructions show you how to compile and link your PLI/VPI C applications so that they can be loaded by ModelSim. Microsoft Visual C/C++ is supported for creating Windows DLLs while gcc and cc compilers are supported for creating UNIX shared libraries.

The PLI/VPI routines are declared in the include files located in the ModelSim *<install_dir>/modeltech/include* directory. The acc_user.h file declares the ACC routines, the veriuser.h file declares the TF routines, and the vpi_user.h file declares the VPI routines.

The following instructions assume that the PLI or VPI application is in a single source file. For multiple source files, compile each file as specified in the instructions and link all of the resulting object files together with the specified link instructions.

Although compilation and simulation switches are platform-specific, loading shared libraries is the same for all platforms. For information on loading libraries, see "Specifying the PLI/VPI file to load" (UM-103).

Windows platforms

```
cl -c -I<install_dir>\modeltech\include app.c
link -dll -export:<init_function> app.obj \
    <install_dir>\modeltech\win32\mtipli.lib /out:app.dll
```

For the Verilog PLI, the <init_function> should be "init_usertfs". Alternatively, if there is no init_usertfs function, the <init_function> specified on the command line should be "veriusertfs". For the Verilog VPI, the <init_function> should be "vlog_startup_routines". These requirements ensure that the appropriate symbol is exported, and thus ModelSim can find the symbol when it dynamically loads the DLL.

The PLI and VPI have been tested with DLLs built using Microsoft Visual C/C++ compiler version 4.1 or greater.

The gcc compiler *cannot* be used to compile PLI/VPI applications under Windows. This is because gcc does not support the Microsoft *.lib/.dll* format.

When executing **cl** commands in a DO file, use the **/NOLOGO** switch to prevent the Microsoft C compiler from writing the logo banner to stderr. Writing the logo causes Tcl to think an error occurred.

Compiling and linking PLI/VPI C++ applications

ModelSim does not have direct support for any language other than standard C; however, C++ code can be loaded and executed under certain conditions.

Since ModelSim's PLI/VPI functions have a standard C prototype, you must prevent the C++ compiler from mangling the PLI/VPI function names. This can be accomplished by using the following type of extern:

```
extern "C"
{
  <PLI/VPI application function prototypes>
}
```

The header files *veriuser.h*, *acc_user.h*, and *vpi_user.h* already include this type of extern. You must also put the PLI/VPI shared library entry point (veriusertfs, init_usertfs, or vlog_startup_routines) inside of this type of extern.

Since ModelSim is a C program and does not include a C++ main, you cannot use iostreams such as cout to print information. You must use io_mcdprintf(), io_printf(), vpi_mcd_printf(), vpi_printf(), vpi_vprintf(), or vpi_mcd_vprintf() to print to the transcript file.

The following platform-specific instructions show you how to compile and link your PLI/VPI C++ applications so that they can be loaded by ModelSim. Microsoft Visual C++ is supported for creating Windows DLLs.

Although compilation and simulation switches are platform-specific, loading shared libraries is the same for all platforms. For information on loading libraries, see "Specifying the PLI/VPI file to load" (UM-103).

Windows platforms

Microsoft Visual C++:

```
cl -c [-GX] -I<install_dir>\modeltech\include app.cxx
link -dll -export:<init_function> app.obj \
     <install_dir>\modeltech\win32\mtipli.lib /out:app.dll
```

The **-GX** argument enables exception handling.

For the Verilog PLI, the **<init_function>** should be "init_usertfs". Alternatively, if there is no init_usertfs function, the **<init_function>** specified on the command line should be "veriusertfs". For the Verilog VPI, the **<init_function>** should be "vlog_startup_routines". These requirements ensure that the appropriate symbol is exported, and thus ModelSim can find the symbol when it dynamically loads the DLL.

The GNU C++ compiler *cannot* be used to compile PLI/VPI applications under Windows. This is because GNU C++ does not support the Microsoft *.lib*/*.dll* format.

When executing **cl** commands in a DO file, use the **/NOLOGO** switch to prevent the Microsoft C compiler from writing the logo banner to stderr. Writing the logo causes Tcl to think an error occurred.

Specifying the PLI/VPI file to load

The PLI/VPI applications are specified as follows:

- As a list in the Veriuser entry in the *modelsim.ini* file:

```
Veriuser = pliapp1.so pliapp2.so pliappn.so
```

- As a list in the PLIOBJS environment variable:

```
% setenv PLIOBJS "pliapp1.so pliapp2.so pliappn.so"
```

- As a **-pli** argument to the simulator (multiple arguments are allowed):

```
-pli pliapp1.so -pli pliapp2.so -pli pliappn.so
```

The various methods of specifying PLI/VPI applications can be used simultaneously. The libraries are loaded in the order listed above. Environment variable references can be used in the paths to the libraries in all cases.

See also *Appendix A - ModelSim variables* for more information on the *modelsim.ini* file.

PLI example

The following example is a trivial, but complete PLI application.

hello.c:

```
#include "veriuser.h"
static PLI_INT32 hello()
{
    io_printf("Hi there\n");
    return 0;
}
s_tfcell veriusertfs[] = {
    {usertask, 0, 0, 0, hello, 0, "$hello"},
    {0}  /* last entry must be 0 */
};
```

hello.v:

```
module hello;
    initial $hello;
endmodule
```

Compile the PLI code for the Solaris operating system:

```
% cc -c -I<install_dir>/modeltech/include hello.c
% ld -G -o hello.sl hello.o
```

Compile the Verilog code:

```
% vlib work
% vlog hello.v
```

Simulate the design:

```
% vsim -c -pli hello.sl hello
# Loading work.hello
# Loading ./hello.sl
VSIM 1> run -all
# Hi there
VSIM 2> quit
```

VPI example

The following example is a trivial, but complete VPI application. A general VPI example can be found in *<install_dir>/modeltech/examples/vpi*.

hello.c:

```
#include "vpi_user.h"
static PLI_INT32 hello(PLI_BYTE8 * param)
{
    vpi_printf( "Hello world!\n" );
    return 0;
}

void RegisterMyTfs( void )
{
    s_vpi_systf_data systf_data;
    vpiHandle systf_handle;
    systf_data.type        = vpiSysTask;
    systf_data.sysfunctype = vpiSysTask;
    systf_data.tfname      = "$hello";
    systf_data.calltf      = hello;
    systf_data.compiletf   = 0;
    systf_data.sizetf      = 0;
    systf_data.user_data   = 0;
    systf_handle = vpi_register_systf( &systf_data );
    vpi_free_object( systf_handle );
}

void (*vlog_startup_routines[])() = {
    RegisterMyTfs,
    0
};
```

hello.v:

```
module hello;
    initial $hello;
endmodule
```

Compile the VPI code for the Solaris operating system:

```
% gcc -c -I<install_dir>/include hello.c
% ld -G -o hello.sl hello.o
```

Compile the Verilog code:

```
% vlib work
% vlog hello.v
```

Simulate the design:

```
% vsim -c -pli hello.sl hello
# Loading work.hello
# Loading ./hello.sl
VSIM 1> run -all
# Hello world!
VSIM 2> quit
```

The PLI callback reason argument

The second argument to a PLI callback function is the reason argument. The values of the various reason constants are defined in the veriuser.h include file. See IEEE Std 1364 for a description of the reason constants. The following details relate to ModelSim Verilog, and may not be obvious in the IEEE Std 1364. Specifically, the simulator passes the reason values to the misctf callback functions under the following circumstances:

`reason_endofcompile`
For the completion of loading the design.

`reason_finish`
For the execution of the $finish system task or the **quit** command.

`reason_startofsave`
For the start of execution of the **checkpoint** command, but before any of the simulation state has been saved. This allows the PLI application to prepare for the save, but it shouldn't save its data with calls to tf_write_save() until it is called with reason_save.

`reason_save`
For the execution of the **checkpoint** command. This is when the PLI application must save its state with calls to tf_write_save().

`reason_startofrestart`
For the start of execution of the **restore** command, but before any of the simulation state has been restored. This allows the PLI application to prepare for the restore, but it shouldn't restore its state with calls to tf_read_restart() until it is called with reason_restart. The reason_startofrestart value is passed only for a restore command, and not in the case that the simulator is invoked with -restore.

`reason_restart`
For the execution of the **restore** command. This is when the PLI application must restore its state with calls to tf_read_restart().

`reason_reset`
For the execution of the **restart** command. This is when the PLI application should free its memory and reset its state. We recommend that all PLI applications reset their internal state during a restart as the shared library containing the PLI code might not be reloaded. (See the **`-keeploaded`** (CR-191) and **`-keeploadedrestart`** (CR-191) arguments to **vsim** for related information.)

`reason_endofreset`
For the completion of the **restart** command, after the simulation state has been reset but before the design has been reloaded.

`reason_interactive`
For the execution of the $stop system task or any other time the simulation is interrupted and waiting for user input.

`reason_scope`
For the execution of the **environment** command or selecting a scope in the structure window. Also for the call to acc_set_interactive_scope() if the callback_flag argument is non-zero.

`reason_paramvc`
For the change of value on the system task or function argument.

`reason_synch`
For the end of time step event scheduled by tf_synchronize().

`reason_rosynch`
For the end of time step event scheduled by tf_rosynchronize().

`reason_reactivate`
For the simulation event scheduled by tf_setdelay().

`reason_paramdrc`
Not supported in ModelSim Verilog.

`reason_force`
Not supported in ModelSim Verilog.

`reason_release`
Not supported in ModelSim Verilog.

`reason_disable`
Not supported in ModelSim Verilog.

The sizetf callback function

A user-defined system function specifies the width of its return value with the sizetf callback function, and the simulator calls this function while loading the design. The following details on the sizetf callback function are not found in the IEEE Std 1364:

- If you omit the sizetf function, then a return width of 32 is assumed.
- The sizetf function should return 0 if the system function return value is of Verilog type "real".
- The sizetf function should return -32 if the system function return value is of Verilog type "integer".

PLI object handles

Many of the object handles returned by the PLI ACC routines are pointers to objects that naturally exist in the simulation data structures, and the handles to these objects are valid throughout the simulation, even after the acc_close() routine is called. However, some of the objects are created on demand, and the handles to these objects become invalid after acc_close() is called. The following object types are created on demand in ModelSim Verilog:

```
accOperator (acc_handle_condition)
accWirePath (acc_handle_path)
accTerminal (acc_handle_terminal, acc_next_cell_load, acc_next_driver, and
    acc_next_load)
accPathTerminal (acc_next_input and acc_next_output)
accTchkTerminal (acc_handle_tchkarg1 and acc_handle_tchkarg2)
accPartSelect (acc_handle_conn, acc_handle_pathin, and acc_handle_pathout)
```

If your PLI application uses these types of objects, then it is important to call acc_close() to free the memory allocated for these objects when the application is done using them.

If your PLI application places value change callbacks on accRegBit or accTerminal objects, *do not* call acc_close() while these callbacks are in effect.

Third party PLI applications

Many third party PLI applications come with instructions on using them with ModelSim Verilog. Even without the instructions, it is still likely that you can get it to work with ModelSim Verilog as long as the application uses standard PLI routines. The following guidelines are for preparing a Verilog-XL PLI application to work with ModelSim Verilog.

Generally, a Verilog-XL PLI application comes with a collection of object files and a veriuser.c file. The veriuser.c file contains the registration information as described above in "Registering PLI applications" (UM-97). To prepare the application for ModelSim Verilog, you must compile the veriuser.c file and link it to the object files to create a dynamically loadable object (see "Compiling and linking PLI/VPI C applications" (UM-101)). For example, if you have a *veriuser.c* file and a library archive *libapp.a* file that contains the application's object files, then the following commands should be used to create a dynamically loadable object for the Solaris operating system:

```
% cc -c -I<install_dir>/modeltech/include veriuser.c
% ld -G -o app.sl veriuser.o libapp.a
```

The PLI application is now ready to be run with ModelSim Verilog. All that's left is to specify the resulting object file to the simulator for loading using the **Veriuser** entry in the *modesim.ini* file, the **-pli** simulator argument, or the PLIOBJS environment variable (see "Registering PLI applications" (UM-97)).

Note: On the HP700 platform, the object files must be compiled as position-independent code by using the **+z** compiler argument. Since, the object files supplied for Verilog-XL may be compiled for static linking, you may not be able to use the object files to create a dynamically loadable object for ModelSim Verilog. In this case, you must get the third party application vendor to supply the object files compiled as position-independent code.

Support for VHDL objects

The PLI ACC routines also provide limited support for VHDL objects in an all VHDL design. The following table lists the VHDL objects for which handles may be obtained and their type and fulltype constants:

Type	Fulltype	Description
accArchitecture	accArchitecture	instantiation of an architecture
accArchitecture	accEntityVitalLevel0	instantiation of an architecture whose entity is marked with the attribute VITAL_Level0
accArchitecture	accArchVitalLevel0	instantiation of an architecture which is marked with the attribute VITAL_Level0
accArchitecture	accArchVitalLevel1	instantiation of an architecture which is marked with the attribute VITAL_Level1
accArchitecture	accForeignArch	instantiation of an architecture which is marked with the attribute FOREIGN and which does not contain any VHDL statements or objects other than ports and generics
accArchitecture	accForeignArchMixed	instantiation of an architecture which is marked with the attribute FOREIGN and which contains some VHDL statements or objects besides ports and generics
accBlock	accBlock	block statement
accForLoop	accForLoop	for loop statement
accForeign	accShadow	foreign scope created by mti_CreateRegion()
accGenerate	accGenerate	generate statement
accPackage	accPackage	package declaration
accSignal	accSignal	signal declaration

The type and fulltype constants for VHDL objects are defined in the *acc_vhdl.h* include file. All of these objects (except signals) are scope objects that define levels of hierarchy in the Structure window. Currently, the PLI ACC interface has no provision for obtaining handles to generics, types, constants, variables, attributes, subprograms, and processes.

IEEE Std 1364 ACC routines

ModelSim Verilog supports the following ACC routines, described in detail in the IEEE Std 1364.

acc_append_delays	acc_append_pulsere	acc_close
acc_collect	acc_compare_handles	acc_configure
acc_count	acc_fetch_argc	acc_fetch_argv
acc_fetch_attribute	acc_fetch_attribute_int	acc_fetch_attribute_str
acc_fetch_defname	acc_fetch_delay_mode	acc_fetch_delays
acc_fetch_direction	acc_fetch_edge	acc_fetch_fullname
acc_fetch_fulltype	acc_fetch_index	acc_fetch_location
acc_fetch_name	acc_fetch_paramtype	acc_fetch_paramval
acc_fetch_polarity	acc_fetch_precision	acc_fetch_pulsere
acc_fetch_range	acc_fetch_size	acc_fetch_tfarg
acc_fetch_itfarg	acc_fetch_tfarg_int	acc_fetch_itfarg_int
acc_fetch_tfarg_str	acc_fetch_itfarg_str	acc_fetch_timescale_info
acc_fetch_type	acc_fetch_type_str	acc_fetch_value
acc_free	acc_handle_by_name	acc_handle_calling_mod_m
acc_handle_condition	acc_handle_conn	acc_handle_hiconn
acc_handle_interactive_scope	acc_handle_loconn	acc_handle_modpath
acc_handle_notifier	acc_handle_object	acc_handle_parent
acc_handle_path	acc_handle_pathin	acc_handle_pathout
acc_handle_port	acc_handle_scope	acc_handle_simulated_net
acc_handle_tchk	acc_handle_tchkarg1	acc_handle_tchkarg2
acc_handle_terminal	acc_handle_tfarg	acc_handle_itfarg
acc_handle_tfinst	acc_initialize	acc_next
acc_next_bit	acc_next_cell	acc_next_cell_load
acc_next_child	acc_next_driver	acc_next_hiconn
acc_next_input	acc_next_load	acc_next_loconn
acc_next_modpath	acc_next_net	acc_next_output
acc_next_parameter	acc_next_port	acc_next_portout

acc_next_primitive	acc_next_scope	acc_next_specparam
acc_next_tchk	acc_next_terminal	acc_next_topmod
acc_object_in_typelist	acc_object_of_type	acc_product_type
acc_product_version	acc_release_object	acc_replace_delays
acc_replace_pulsere	acc_reset_buffer	acc_set_interactive_scope
acc_set_pulsere	acc_set_scope	acc_set_value
acc_vcl_add	acc_vcl_delete	acc_version

Note: acc_fetch_paramval() cannot be used on 64-bit platforms to fetch a string value of a parameter. Because of this, the function acc_fetch_paramval_str() has been added to the PLI for this use. acc_fetch_paramval_str() is declared in acc_user.h. It functions in a manner similar to acc_fetch_paramval() except that it returns a char *. acc_fetch_paramval_str() can be used on all platforms.

IEEE Std 1364 TF routines

ModelSim Verilog supports the following TF routines, described in detail in the IEEE Std 1364.

io_mcdprintf	io_printf	mc_scan_plusargs
tf_add_long	tf_asynchoff	tf_iasynchoff
tf_asynchon	tf_iasynchon	tf_clearalldelays
tf_iclearalldelays	tf_compare_long	tf_copypvc_flag
tf_icopypvc_flag	tf_divide_long	tf_dofinish
tf_dostop	tf_error	tf_evaluatep
tf_ievaluatep	tf_exprinfo	tf_iexprinfo
tf_getcstringp	tf_igetcstringp	tf_getinstance
tf_getlongp	tf_igetlongp	tf_getlongtime
tf_igetlongtime	tf_getnextlongtime	tf_getp
tf_igetp	tf_getpchange	tf_igetpchange
tf_getrealp	tf_igetrealp	tf_getrealtime
tf_igetrealtime	tf_gettime	tf_igettime
tf_gettimeprecision	tf_igettimeprecision	tf_gettimeunit
tf_igettimeunit	tf_getworkarea	tf_igetworkarea

tf_long_to_real	tf_longtime_tostr	tf_message
tf_mipname	tf_imipname	tf_movepvc_flag
tf_imovepvc_flag	tf_multiply_long	tf_nodeinfo
tf_inodeinfo	tf_nump	tf_inump
tf_propagatep	tf_ipropagatep	tf_putlongp
tf_iputlongp	tf_putp	tf_iputp
tf_putrealp	tf_iputrealp	tf_read_restart
tf_real_to_long	tf_rosynchronize	tf_irosynchronize
tf_scale_longdelay	tf_scale_realdelay	tf_setdelay
tf_isetdelay	tf_setlongdelay	tf_isetlongdelay
tf_setrealdelay	tf_isetrealdelay	tf_setworkarea
tf_isetworkarea	tf_sizep	tf_isizep
tf_spname	tf_ispname	tf_strdelputp
tf_istrdelputp	tf_strgetp	tf_istrgetp
tf_strgettime	tf_strlongdelputp	tf_istrlongdelputp
tf_strrealdelputp	tf_istrrealdelputp	tf_subtract_long
tf_synchronize	tf_isynchronize	tf_testpvc_flag
tf_itestpvc_flag	tf_text	tf_typep
tf_itypep	tf_unscale_longdelay	tf_unscale_realdelay
tf_warning	tf_write_save	

Verilog-XL compatible routines

The following PLI routines are not defined in IEEE Std 1364, but ModelSim Verilog provides them for compatibility with Verilog-XL.

```
char *acc_decompile_exp(handle condition)
```

This routine provides similar functionality to the Verilog-XL **acc_decompile_expr** routine. The condition argument must be a handle obtained from the acc_handle_condition routine. The value returned by **acc_decompile_exp** is the string representation of the condition expression.

```
char *tf_dumpfilename(void)
```

This routine returns the name of the VCD file.

```
void tf_dumpflush(void)
```

A call to this routine flushes the VCD file buffer (same effect as calling **$dumpflush** in the Verilog code).

```
int tf_getlongsimtime(int *aof_hightime)
```

This routine gets the current simulation time as a 64-bit integer. The low-order bits are returned by the routine, while the high-order bits are stored in the aof_hightime argument.

64-bit support in the PLI

The PLI function acc_fetch_paramval() cannot be used on 64-bit platforms to fetch a string value of a parameter. Because of this, the function acc_fetch_paramval_str() has been added to the PLI for this use. acc_fetch_paramval_str() is declared in acc_user.h. It functions in a manner similar to acc_fetch_paramval() except that it returns a char *. acc_fetch_paramval_str() can be used on all platforms.

PLI/VPI tracing

The foreign interface tracing feature is available for tracing PLI and VPI function calls. Foreign interface tracing creates two kinds of traces: a human-readable log of what functions were called, the value of the arguments, and the results returned; and a set of C-language files that can be used to replay what the foreign interface code did.

The purpose of tracing files

The purpose of the logfile is to aid you in debugging PLI or VPI code. The primary purpose of the replay facility is to send the replay files to MTI support for debugging co-simulation problems, or debugging PLI/VPI problems for which it is impractical to send thePLI/VPI code. We still need you to send the VHDL/Verilog part of the design to actually execute a replay, but many problems can be resolved with the trace only.

Invoking a trace

To invoke the trace, call **vsim** (CR-189) with the **-trace_foreign** argument:

Syntax

```
vsim
  -trace_foreign <action> [-tag <name>]
```

Arguments

`<action>`
Specifies one of the following actions:

Value	Action	Result
1	create log only	writes a local file called "mti_trace_<tag>"
2	create replay only	writes local files called "mti_data_<tag>.c", "mti_init_<tag>.c", "mti_replay_<tag>.c" and "mti_top_<tag>.c"
3	create both log and replay	

`-tag <name>`
Used to give distinct file names for multiple traces. Optional.

Examples

```
vsim -trace_foreign 1 mydesign
```
Creates a logfile.

```
vsim -trace_foreign 3 mydesign
```
Creates both a logfile and a set of replay files.

```
vsim -trace_foreign 1 -tag 2 mydesign
```
Creates a logfile with a tag of "2".

The tracing operations will provide tracing during all user foreign code-calls, includingPLI/VPI user tasks and functions (calltf, checktf, sizetf and misctf routines), and Verilog VCL callbacks.

Debugging PLI/VPI application code

In order to debug your PLI/VPI application code in a debugger, your application code must be compiled with debugging information (for example, by using the **-g** option) and without optimizations (for example, don't use the **-O** option). You must then load **vsim** into a debugger. Even though **vsim** is stripped, most debuggers will still execute it. You can invoke the debugger directly on **vsim** (for example, "`ddd 'which vsim'`") or you can attach the debugger to an already running **vsim** process. In the second case, you must attach to the PID for **vsim**, and you must specify the full path to the **vsim** executable (for example, "`gdb $MTI_HOME/sunos5/vsim 1234`").

On Solaris, AIX, and Linux systems you can use either **gdb** or **ddd**. On HP-UX systems you can use the **wdb** debugger from HP. You will need version 1.2 or later.

Since initially the debugger recognizes only **vsim's** PLI/VPI function symbols, when invoking the debugger directly on **vsim** you need to place a breakpoint in the first PLI/VPI function that is called by your application code. An easy way to set an entry point is to put a call to acc_product_version() as the first executable statement in your application code. Then, after **vsim** has been loaded into the debugger, set a breakpoint in this function. Once you have set the breakpoint, run vsim with the usual arguments (e.g., "run -c top").

On HP-UX you might see some warning messages that **vsim** does not have debugging information available. This is normal. If you are using Exceed to access an HP machine from Windows NT, it is recommended that you run **vsim** in command line or batch mode because your NT machine may hang if you run **vsim** in GUI mode. Click on the "go" button, or use F5 or the **go** command to execute **vsim** in **wdb**.

When the breakpoint is reached, the shared library containing your application code has been loaded. In some debuggers you must use the **share** command to load the PLI/VPI application's symbols.

On HP-UX you might see a warning about not finding "__dld_flags" in the object file. This warning can be ignored. You should see a list of libraries loaded into the debugger. It should include the library for your PLI/VPI application. Alternatively, you can use **share** to load only a single library.

At this point all of the PLI/VPI application's symbols should be visible. You can now set breakpoints in and single step through your PLI/VPI application code.

6 - WLF files (datasets) and virtuals

Chapter contents

WLF files (datasets) UM-118
Saving a simulation to a WLF file UM-119
Opening datasets UM-119
Viewing dataset structure UM-120
Managing multiple datasets UM-121
Saving at intervals with Dataset Snapshot UM-123
Virtual Objects (User-defined buses, and more) UM-125

Virtual Objects (User-defined buses, and more) UM-125
Virtual signals UM-125
Virtual functions UM-126
Virtual regions UM-127
Virtual types UM-127

Dataset, WLF file, and virtual commands UM-128

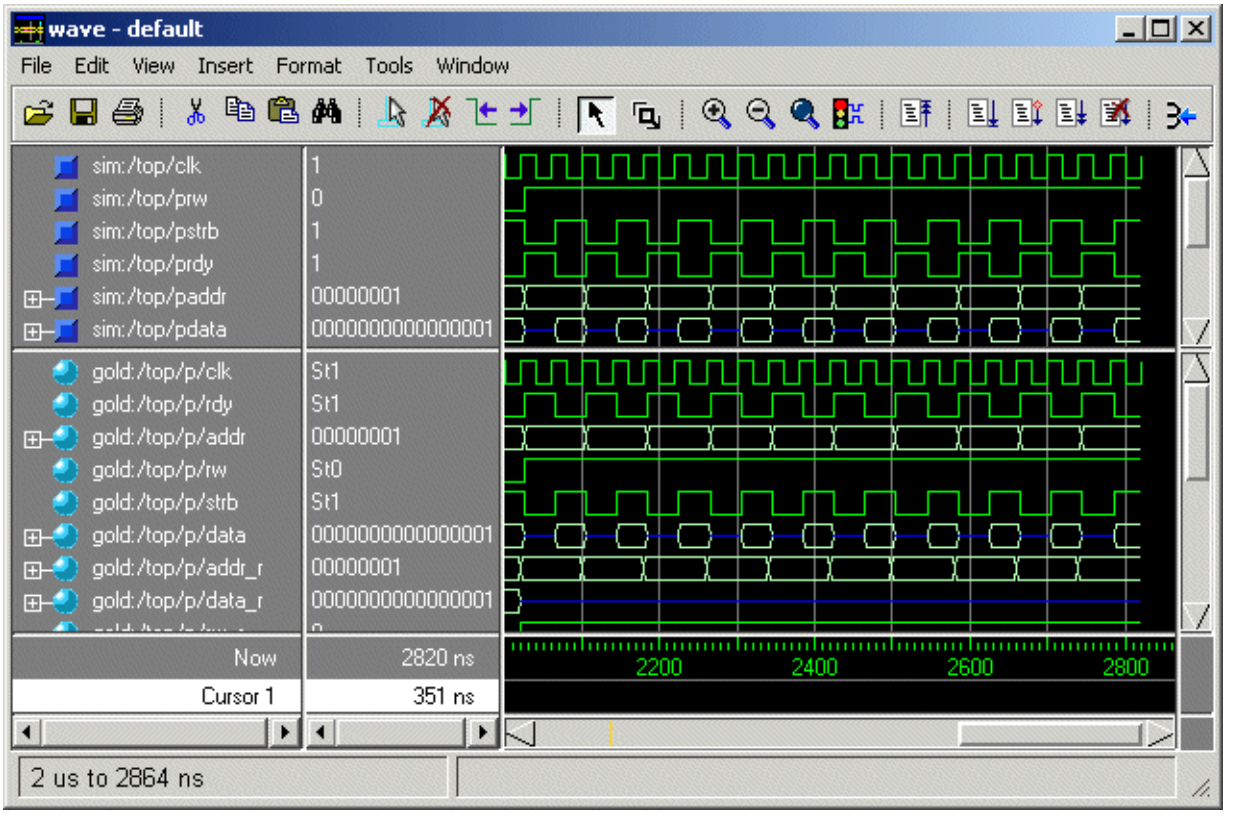
A ModelSim simulation can be saved to a wave log format (WLF) file for future viewing or comparison to a current simulation. We use the term "dataset" to refer to a WLF file that has been reopened for viewing.

With ModelSim release 5.3 and later, you can open more than one WLF file for simultaneous viewing. You can also create virtual signals that are simple logical combinations of, or logical functions of, signals from different datasets.

WLF files (datasets)

Wave log format (WLF) files store saved simulation data. Any number of WLF files can be reloaded for viewing or comparing to the active simulation. The term "dataset" refers to a logical name that is assigned to the WLF file when it is reloaded.

A dataset prefix identifies each WLF file that is opened. The current active simulation is prefixed by "sim," while any datasets are prefixed by the name of the WLF file. For example, two datasets are displayed in the Wave window below—the current simulation is shown in the top pane and is indicated by the "sim" prefix; a dataset from a previous simulation is shown in the bottom pane and is indicated by the "gold" prefix.



Note: The simulator resolution (see "Simulator resolution limit" (UM-52)) must be the same for all datasets you're comparing, including the current simulation.

Saving a simulation to a WLF file

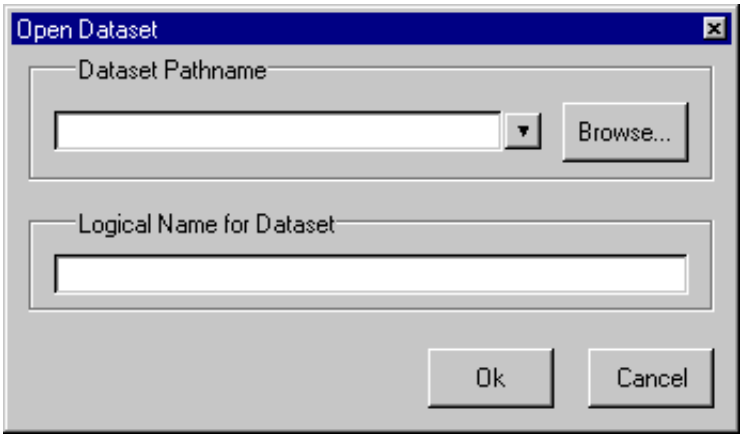
If you add items to the Dataflow, List, or Wave windows, or log items with the **log** command, the results of each simulation run are automatically saved to a WLF file called *vsim.wlf* in the current directory. If you run a new simulation in the same directory, the *vsim.wlf* file is overwritten with the new results.

If you want to save the WLF file and not have it overwritten, select **File > Save Dataset > sim** (Main window) or **File > Save> sim dataset** (Wave window). Or, you can use the **-wlf <filename>** argument to the **vsim** command (CR-189) or the **dataset save** command (CR-62).

Important: If you do not use **dataset save** or **dataset snapshot**, you must end a simulation session with a **quit** or **quit -sim** command in order to produce a valid WLF file. If you don't end the simulation in this manner, the WLF file will not close properly. ModelSim may issue the error message "bad magic number" when you try to open an incomplete dataset in subsequent sessions.

Opening datasets

To open a dataset, select either **File > Open > Dataset** (Main window) or use the **dataset open** command (CR-60).



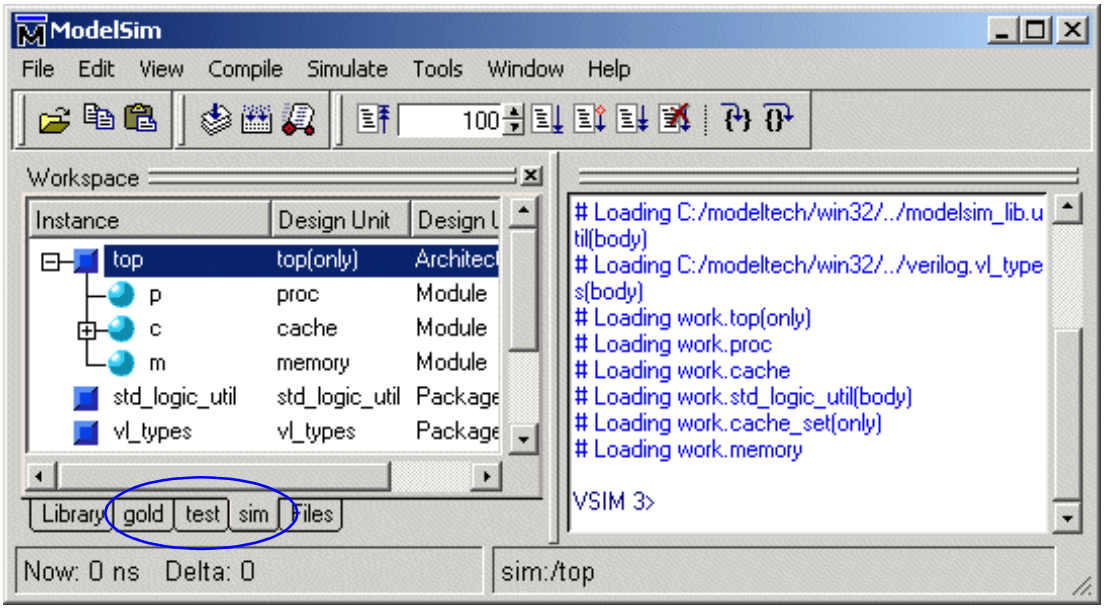
The Open Dataset dialog includes the following options.

- **Dataset Pathname**
  Identifies the path and filename of the WLF file you want to open.
- **Logical Name for Dataset**
  This is the name by which the dataset will be referred. By default this is the name of the WLF file.

Viewing dataset structure

Each dataset you open creates a Structure tab in the Main window workspace. The tab is labeled with the name of the dataset and displays the same data as the "Structure window" (UM-199).

The graphic below shows three Structure tabs: one for the active simulation (*sim*) and one each for two datasets (*gold* and *test*).



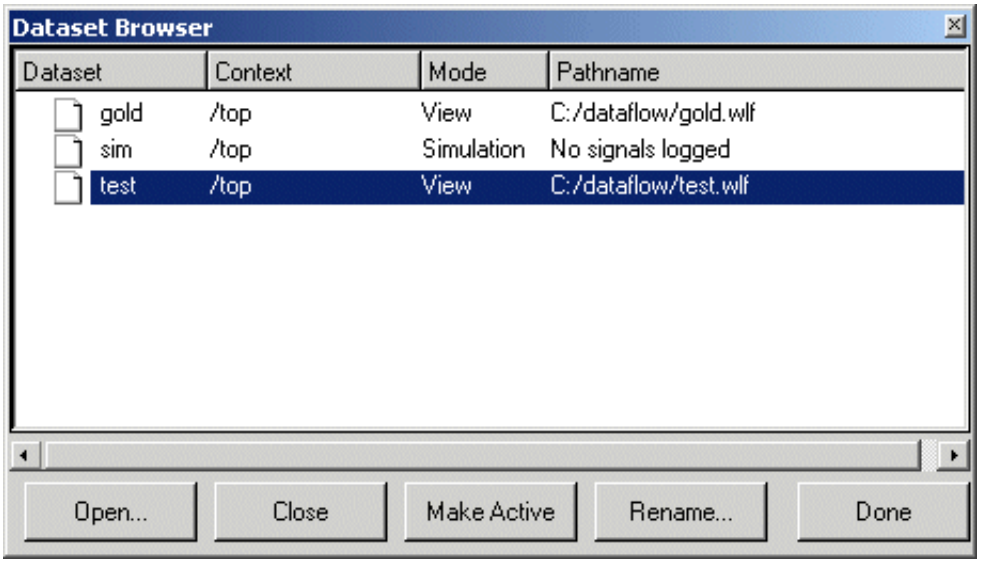
If you have too many tabs to display in the available space, you can scroll the tabs left or right by clicking and dragging them .

Each Structure tab has a context menu that you access by clicking the right mouse button. See "Structure window context menu" (UM-201) for details.

Managing multiple datasets

GUI

When you have one or more datasets open, you can manage them using the **Dataset Browser**. To open the browser, select **View > Datasets** (Main window).



The Dataset Browser dialog box includes the following options.

- **Open**
  Opens the Open Dataset dialog box (see "Opening datasets" (UM-119)) so you can open additional datasets.
- **Close**
  Closes the selected dataset. This will also remove the dataset's Structure tab in the Main window workspace.
- **Make Active**
  Makes the selected dataset "active." You can also effect this change by double-clicking the dataset name. Active dataset means that if you type a region path as part of a command and omit the dataset prefix, the active dataset will be assumed. It is equivalent to typing `env <dataset>:` at the VSIM prompt. The active dataset is displayed at the bottom of the Main window.
- **Rename**
  Allows you to assign a new logical name for the selected dataset.

Command line

You can open multiple datasets when the simulator is invoked by specifying more than one **vsim -view <filename>** option. By default the dataset prefix will be the filename of the WLF file. You can specify a different dataset name as an optional qualifier to the **vsim -view** switch on the command line using the following syntax:

```
-view <dataset>=<filename>
```

For example: **vsim -view foo=vsim.wlf**

ModelSim designates one of the datasets to be the "active" dataset, and refers all names without dataset prefixes to that dataset. The active dataset is displayed in the context path at the bottom of the Main window. When you select a design unit in a dataset's Structure tab, that dataset becomes active automatically. Alternatively, you can use the Dataset Browser or the **environment** command (CR-74) to change the active dataset.

Design regions and signal names can be fully specified over multiple WLF files by using the dataset name as a prefix in the path. For example:

```
sim:/top/alu/out
view:/top/alu/out
golden:.top.alu.out
```

Dataset prefixes are not required unless more than one dataset is open, and you want to refer to something outside the active dataset. When more than one dataset is open, ModelSim will automatically prefix names in the Wave and List windows with the dataset name. You can change this default by selecting **Tools > Window Preferences** (Wave and List windows).

ModelSim also remembers a "current context" within each open dataset. You can toggle between the current context of each dataset using the **environment** command (CR-74), specifying the dataset without a path. For example:

```
env foo:
```

sets the active dataset to **foo** and the current context to the context last specified for **foo**. The context is then applied to any unlocked windows.

The current context of the current dataset (usually referred to as just "current context") is used for finding objects specified without a path.

The Signals window can be locked to a specific context of a dataset. Being locked to a dataset means that the window will update only when the content of that dataset changes. If locked to both a dataset and a context (e.g., test: /top/foo), the window will update only when that specific context changes. You specify the dataset to which the window is locked by selecting **File > Environment** (Signals window).

Restricting the dataset prefix display

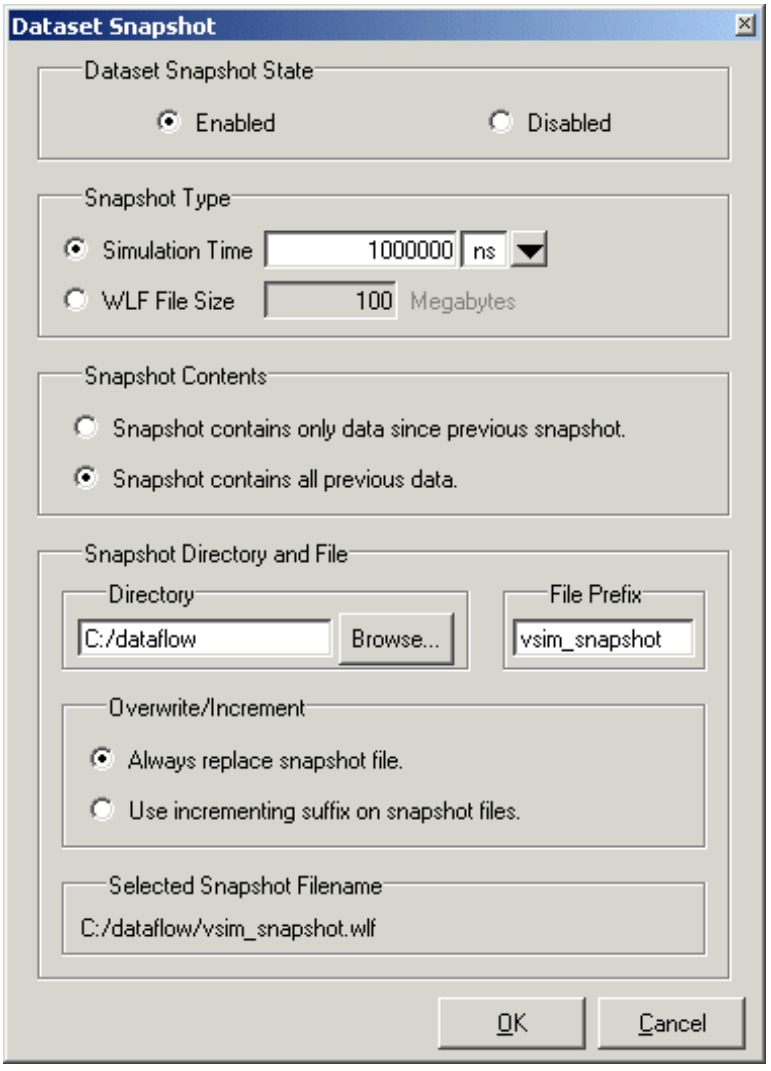
The default for dataset prefix viewing is set with a variable in *pref.tcl*, **PrefMain(DisplayDatasetPrefix)**. Setting the variable to 1 will display the prefix, setting it to 0 will not. It is set to 1 by default. Either edit the *pref.tcl* file directly or use the **Tools > Edit Preferences** (Main window) command to change the variable value.

Additionally, you can restrict display of the dataset prefix if you use the **environment -nodataset** command to view a dataset. To display the prefix use the **environment** command (CR-74) with the **-dataset** option (you won't need to specify this option if the variable noted above is set to 1). The **environment** command line switches override the *pref.tcl* variable.

Saving at intervals with Dataset Snapshot

Dataset Snapshot lets you periodically copy data from the current simulation WLF file to another file. This is useful for taking periodic "snapshots" of your simulation or for clearing the current simulation WLF file based on size or elapsed time.

Once you have logged the appropriate items, select **Tools** > **Dataset Snapshot** (Wave window).



The Dataset Snapshot dialog includes these options:

Dataset Snapshot State

- **Enabled/Disabled**
  Enable or disable Dataset Snapshot. All other dialog options are unavailable if Disabled is selected.

Snapshot Type

- **Simulation Time**
  Specifies that data is copied to the specified snapshot file every <x> time units. Default is 1000000 time units.
- **WLF File Size**
  Specifies that data is copied to the specified snapshot file whenever the current simulation WLF file reaches <*x*> megabytes. Default is 100 MB.

Snapshot Contents

- **Snapshot contains only data since previous snapshot**
  Specifies that each snapshot contains only data since the last snapshot. This option causes ModelSim to clear the current simulation WLF file each time a snapshot is taken.
- **Snapshot contains all previous data**
  Specifies that each snapshot contains all data from the time signals were first logged. The entire contents of the current simulation WLF file are saved each time a snapshot is taken.

Snapshot Directory and File

- **Directory**
  The directory in which ModelSim saves the snapshot files.
- **File Prefix**
  The name of the snapshot files. ModelSim adds *.wlf* to the snapshot files.

Overwrite / Increment

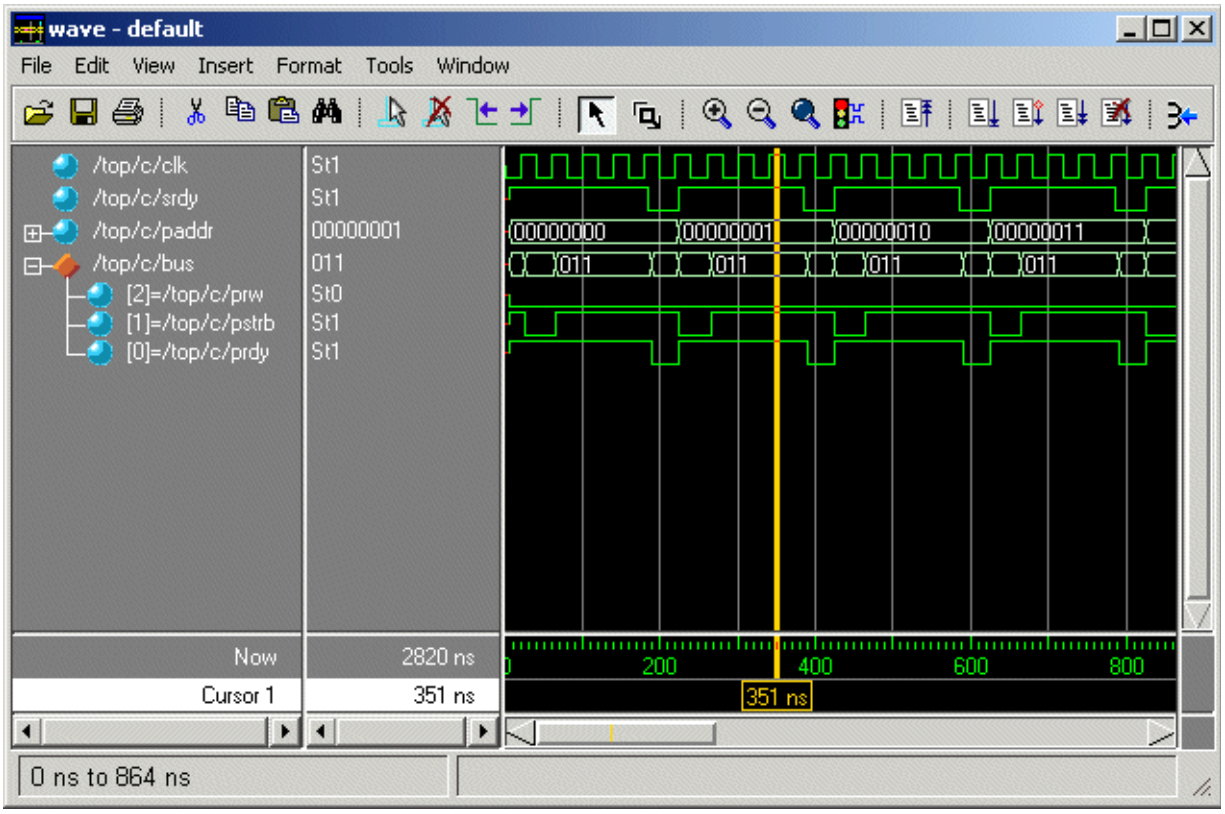
- **Always replace snapshot file**
  Specifies that a single file is created for all snapshots. Each new snapshot overwrites the previous.
- **Use incrementing suffix on snapshot files**
  Specifies that a new file is created for each snapshot. Each new snapshot creates a separate file (e.g., *vsim_snapshot_0.wlf*, *vsim_snapshot_1.wlf*, etc.).

Virtual Objects (User-defined buses, and more)

Virtual objects are signal-like or region-like objects created in the GUI that do not exist in the ModelSim simulation kernel. ModelSim supports the following kinds of virtual objects:

- Virtual signals (UM-125)
- Virtual functions (UM-126)
- Virtual regions (UM-127)
- Virtual types (UM-127)

Virtual objects are indicated by an orange diamond as illustrated by *bus* below:



Virtual signals

Virtual signals are aliases for combinations or subelements of signals written to the WLF file by the simulation kernel. They can be displayed in the Signals, List, and Wave windows, accessed by the **examine** command, and set using the **force** command. You can create virtual signals using the **Tools > Combine Signals** (Wave and List windows) command or use the **virtual signal** command (CR-175). Once created, virtual signals can be dragged and dropped from the Signals window to the Wave and List windows.

Virtual signals are automatically attached to the design region in the hierarchy that corresponds to the nearest common ancestor of all the elements of the virtual signal. The **virtual signal** command has an **-install <region>** option to specify where the virtual signal should be installed. This can be used to install the virtual signal in a user-defined region in

order to reconstruct the original RTL hierarchy when simulating and driving a post-synthesis, gate-level implementation.

A virtual signal can be used to reconstruct RTL-level design buses that were broken down during synthesis. The **virtual hide** command (CR-166) can be used to hide the display of the broken-down bits if you don't want them cluttering up the Signals window.

If the virtual signal has elements from more than one WLF file, it will be automatically installed in the virtual region *virtuals:/Signals*.

Virtual signals are not hierarchical – if two virtual signals are concatenated to become a third virtual signal, the resulting virtual signal will be a concatenation of all the subelements of the first two virtual signals.

The definitions of virtuals can be saved to a macro file using the **virtual save** command (CR-173). By default, when quitting, ModelSim will append any newly-created virtuals (that have not been saved) to the *virtuals.do* file in the local directory.

If you have virtual signals displayed in the Wave or List window when you save the Wave or List format, you will need to execute the *virtuals.do* file (or some other equivalent) to restore the virtual signal definitions before you re-load the Wave or List format during a later run. There is one exception: "implicit virtuals" are automatically saved with the Wave or List format.

Implicit and explicit virtuals

An implicit virtual is a virtual signal that was automatically created by ModelSim without your knowledge and without you providing a name for it. An example would be if you expand a bus in the Wave window, then drag one bit out of the bus to display it separately. That action creates a one-bit virtual signal whose definition is stored in a special location, and is not visible in the Signals window or to the normal virtual commands.

All other virtual signals are considered "explicit virtuals".

Virtual functions

Virtual functions behave in the GUI like signals but are not aliases of combinations or elements of signals logged by the kernel. They consist of logical operations on logged signals and can be dependent on simulation time. They can be displayed in the Signals, Wave, and List windows and accessed by the **examine** command (CR-75), but cannot be set by the **force** command (CR-82).

Examples of virtual functions include the following:

- a function defined as the inverse of a given signal
- a function defined as the exclusive-OR of two signals
- a function defined as a repetitive clock
- a function defined as "the rising edge of CLK delayed by 1.34 ns"

Virtual functions can also be used to convert signal types and map signal values.

The result type of a virtual signal can be any of the types supported in the GUI expression syntax: integer, real, boolean, std_logic, std_logic_vector, and arrays and records of these types. Verilog types are converted to VHDL 9-state std_logic equivalents and Verilog net strengths are ignored.

Virtual functions can be created using the **virtual function** command (CR-163).

Virtual functions are also implicitly created by ModelSim when referencing bit-selects or part-selects of Verilog registers in the GUI, or when expanding Verilog registers in the Signals, Wave or List window. This is necessary because referencing Verilog register elements requires an intermediate step of shifting and masking of the Verilog "vreg" data structure.

Virtual regions

User-defined design hierarchy regions can be defined and attached to any existing design region or to the virtuals context tree. They can be used to reconstruct the RTL hierarchy in a gate-level design and to locate virtual signals. Thus, virtual signals and virtual regions can be used in a gate-level design to allow you to use the RTL test bench.

Virtual regions are created and attached using the **virtual region** command (CR-172).

Virtual types

User-defined enumerated types can be defined in order to display signal bit sequences as meaningful alphanumeric names. The virtual type is then used in a type conversion expression to convert a signal to values of the new type. When the converted signal is displayed in any of the windows, the value will be displayed as the enumeration string corresponding to the value of the original signal.

Virtual types are created using the **virtual type** command (CR-178).

Dataset, WLF file, and virtual commands

The table below provides a brief description of the actions associated with datasets, WLF files, and virtual commands. For complete details about syntax, arguments, and usage, refer to the *ModelSim Command Reference*.

Command name	Action
dataset alias (CR-55)	assigns an additional name (alias) to a dataset
dataset clear (CR-56)	removes all event data from the current simulation WLF file while keeping all currently logged signals logged
dataset close (CR-57)	closes the specified dataset
dataset info (CR-58)	reports a variety of information about a dataset
dataset list (CR-59)	lists all open datasets
dataset open (CR-60)	opens a WLF file
dataset rename (CR-61)	assigns a new logical name to the specified dataset
dataset save (CR-62)	saves the current simulation to a WLF file
dataset snapshot (CR-63)	saves the current simulation to a WLF file at regular intervals
log (CR-87)	creates a WLF file for the current simulation
nolog (CR-93)	suspends writing of data to the WLF file for the specified signals
searchlog (CR-116)	searches one or more of the currently open WLF files for a specified condition
virtual function (CR-163)	creates a new signal that consists of logical operations on existing signals and simulation time
virtual region (CR-172)	creates a new user-defined design hierarchy region
virtual signal (CR-175)	creates a new signal that consists of concatenations of signals and subelements
virtual type (CR-178)	creates a new enumerated type
vsim (CR-189) -wlf <filename>	creates a WLF file for the simulation which can be reopened as a dataset
wlf2log (CR-211)	translates a ModelSim WLF file (*vsim.wlf*) to a QuickSim II logfile
wlfman (CR-213)	allows you to get information about and manipulate WLF files
wlfrecover (CR-215)	attempts to "repair" WLF files that are incomplete due to a crash or the file being copied prior to completion of the simulation

7 - Graphic interface

Chapter contents

Window overview UM-130
Common window features UM-131
Main window UM-137
Dataflow window UM-149
List window UM-168
Process window UM-181
Signals window UM-183
Source window UM-191
Structure window UM-199
Variables window UM-203
Wave window UM-206
Compiling with the graphic interface UM-238
Simulating with the graphic interface UM-245
Creating and managing breakpoints UM-258
Miscellaneous tools and add-ons UM-262
Graphic interface commands UM-267

Window overview

The ModelSim simulation and debugging environment consists of nine windows. A brief description of each window follows:

- Main window (UM-137)
  The initial window that appears upon startup. All subsequent ModelSim windows are opened from the Main window. This window contains the session transcript; the Workspace, which can contain Project, Library, Structure, and Files tabs; and the coverage panes when you have simulated with "Code Coverage" (UM-283).
- Dataflow window (UM-149)
  Lets you trace signals and nets through your design by showing related processes.
- List window (UM-168)
  Shows the simulation values of selected VHDL signals and variables and Verilog nets, registers, and variables in tabular format.
- Process window (UM-181)
  Displays a list of processes in the region currently selected in the Structure window.
- Signals window (UM-183)
  Shows the names and current values of VHDL signals, and Verilog nets, registers, and variables in the region currently selected in the Structure window.
- Source window (UM-191)
  Displays the HDL source code for the design.
- Structure window (UM-199)
  Displays the hierarchy of structural elements such as VHDL component instances, packages, blocks, generate statements, and Verilog model instances, named blocks, tasks and functions. In versions 5.5 and later, this same information is displayed in the Main window workspace.
- Variables window (UM-203)
  Displays VHDL constants, generics, variables, and Verilog registers and variables in the current process and their current values.
- Wave window (UM-206)
  Displays waveforms, and current values for the VHDL signals and variables and Verilog nets, registers, and variables you have selected. Current and past simulations can be compared side-by-side in one Wave window.

Common window features

ModelSim's graphic interface provides many features that add to its usability; features common to many of the windows are described below.

Feature	Feature applies to these windows
Quick access toolbars (UM-132)	Dataflow, Main, Source, and Wave windows
Drag and Drop (UM-132)	Dataflow, List, Process, Signals, Source, Structure, Variables, and Wave windows
Command history (UM-132)	Main window command line
Automatic window updating (UM-133)	Dataflow, Process, Signals, and Structure windows
Finding names (UM-133)	various windows
Sorting HDL items (UM-133)	Process, Signals, Source, Structure, Variables and Wave windows
Menu tear off (UM-134)	all windows
Combining items in the List window (UM-174), Combining items in the Wave window (UM-217)	List and Wave windows
Tree window hierarchical view (UM-135)	Structure, Signals, Variables, and Wave windows

- Cut/Copy/Paste/Delete into any entry box by clicking the right mouse button in the entry box.
- Standard cut/copy/paste shortcut keystrokes – ^X/^C/^V – will work in all entry boxes.
- When the focus changes to an entry box, the contents of that box are selected (highlighted). This allows you to replace the current contents of the entry box with new contents with a simple paste command, without having to delete the old value.
- Dialog boxes will appear on top of their parent window (instead of the upper left corner of the screen).
- You can change the title of any window with the -title switch of the **view** command. See **view** command (CR-156) for details.

- The middle mouse button will allow you to paste the following into the transcript window:

  –text currently selected in the transcript window,

  –a current primary X-Windows selection (can be from another application), or

  –contents of the clipboard.

Note: Selecting text in the transcript window makes it the current primary X-Windows selection. This way you can copy transcript window selections to other X-Windows windows (xterm, emacs, etc.).

- The **Edit > Paste** operation in the Transcript pane will ONLY paste from the clipboard.
- All menus highlight their accelerator keys.

Quick access toolbars

Buttons on the Dataflow, Main, Source, and Wave windows provide access to commonly used commands and functions.

Drag and Drop

Drag and drop of HDL items is possible between the following windows. Using the left mouse button, click and release to select an item, then click and hold to drag it.

- **Drag items from these windows:**
  Dataflow, List, Process, Signals, Source, Structure, Variables, and Wave windows
- **Drop items into these windows:**
  Dataflow, List, and Wave windows

Note: Drag and drop works to rearrange items *within* the List and Wave windows as well.

Command history

Avoid entering long commands twice; use the down and up keyboard arrows to move through the command history for the current simulation.

Automatic window updating

Selecting an item in the following windows automatically updates other related ModelSim windows as indicated below:

Select an item in this window	To update these windows
Dataflow window (UM-149)	Process window (UM-181)
	Signals window (UM-183)
	Source window (UM-191)
	Structure window (UM-199)
	Variables window (UM-203)
Process window (UM-181)	Dataflow window (UM-149)
	Signals window (UM-183)
	Source window (UM-191)
	Variables window (UM-203)
Signals window (UM-183)	Dataflow window (UM-149)
Structure window (UM-199) or structure pane in Main window Workspace	Process window (UM-181)
	Signals window (UM-183)
	Source window (UM-191)

Finding names

- **Find** HDL item names with the **Edit > Find** menu selection in these windows: Dataflow, List, Process, Signals, Source, Structure, Variables, and Wave windows.

  A **Find** request that starts with a backslash (\ ) forces case sensitivity. Elsewhere in the pattern backslashes are used to escape special interpretation of basic regular expression characters. To search explicitly for a backslash character, it is necessary to escape the character. For example, to match \Arch Signal 1\, the pattern \\Arch... is required.

Sorting HDL items

Use the **View > Sort** menu selection in the Process, Signals, Structure, Variables and Wave windows to sort HDL items in ascending, descending or declaration order.

Names such as *net_1*, *net_10*, and *net_2* will sort numerically in the Signals and Wave windows.

Saving window layout

You can save the current positions and sizes of ModelSim windows as a default. Follow these steps to save the layout as a default:

1. Position and size the windows the way you want them to display.
2. Select **Tools > Save Preferences** (Main window) and save the *modelsim.tcl* file into the desired directory.
3. Modify the "Working Directory" of your ModelSim shortcut to point at the directory, or set the MODELSIM_TCL environment variable to point at the directory (see "Creating environment variables in Windows" (UM-339) for more details).

Context menus

Context menus refer to menus that "pop-up" in the middle of the interface by clicking the right mouse button. The commands on the menu change depending on where in the interface you click. In other words, the menus change based on the context of their use. These menus are available in the following windows: Dataflow, List, Main, Signals, Source, Structure, and Wave.

Menu tear off

All window menus can be "torn off " to create a separate menu window. To tear off, click on the menu, then select the dotted-line button at the top of the menu.

Tree window hierarchical view

ModelSim provides a hierarchical, or "tree view" of some aspects of your design in the Main window Structure tabs and the Structure, Signals, Variables, and Wave windows.

HDL items you can view

Depending on which window you are viewing, one entry is created for each of the following VHDL and Verilog HDL items within the design:

VHDL items

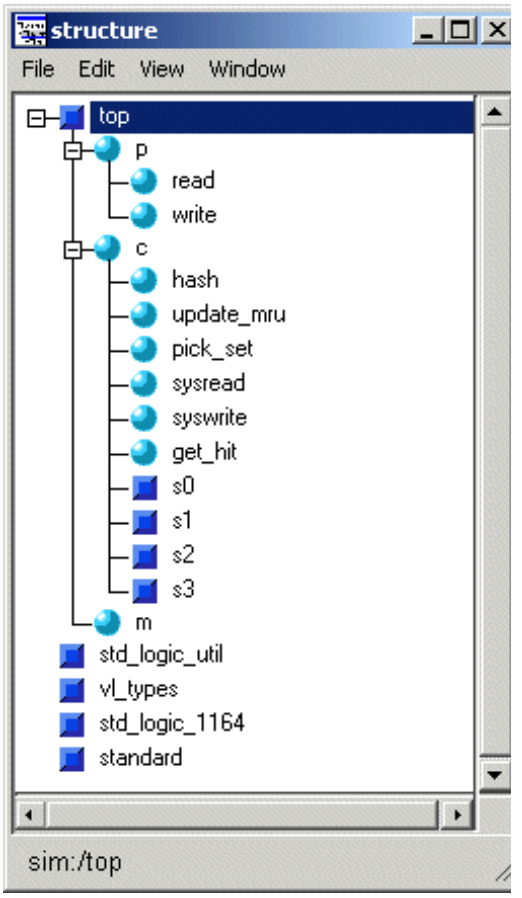
(indicated by a dark blue square icon) signals, variables, component instantiations, generate statements, block statements, and packages

Verilog items

(indicated by a lighter blue circle icon) parameters, registers, nets, module instantiations, named forks, named begins, tasks, and functions

Virtual items

(indicated by an orange diamond icon) virtual signals, buses, and functions, see "Virtual Objects (User-defined buses, and more)" (UM-125) for more information



Viewing the hierarchy

Whenever you see a tree view, as in the Structure window displayed here, you can use the mouse to collapse or expand the hierarchy. Select the symbols as shown below to change the view of the structure.

Symbol	Description
[+]	click a plus box to expand the item and view the structure
[-]	click a minus box to hide a hierarchy that has been expanded

Finding items within tree windows

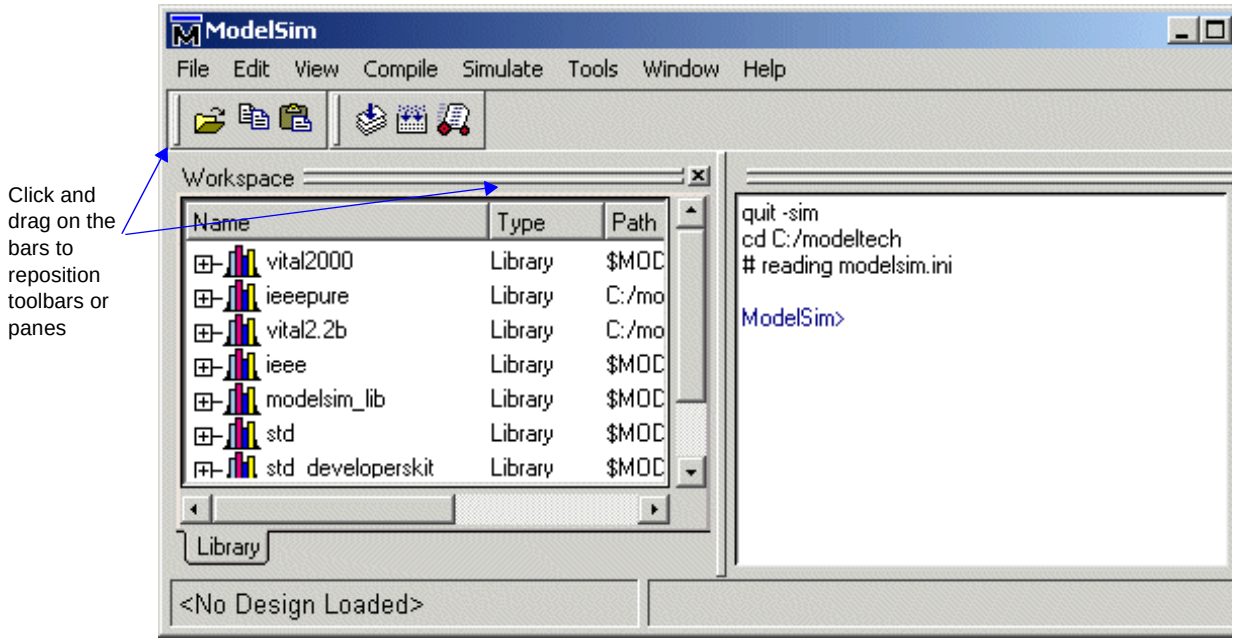
You can open the Find dialog box within all windows by selecting **Edit > Find** or by using **<control-s>** (UNIX) or **<control-f>** (Windows).

Options within the Find dialog box allow you to search unique text-string fields within the specific window. See also,

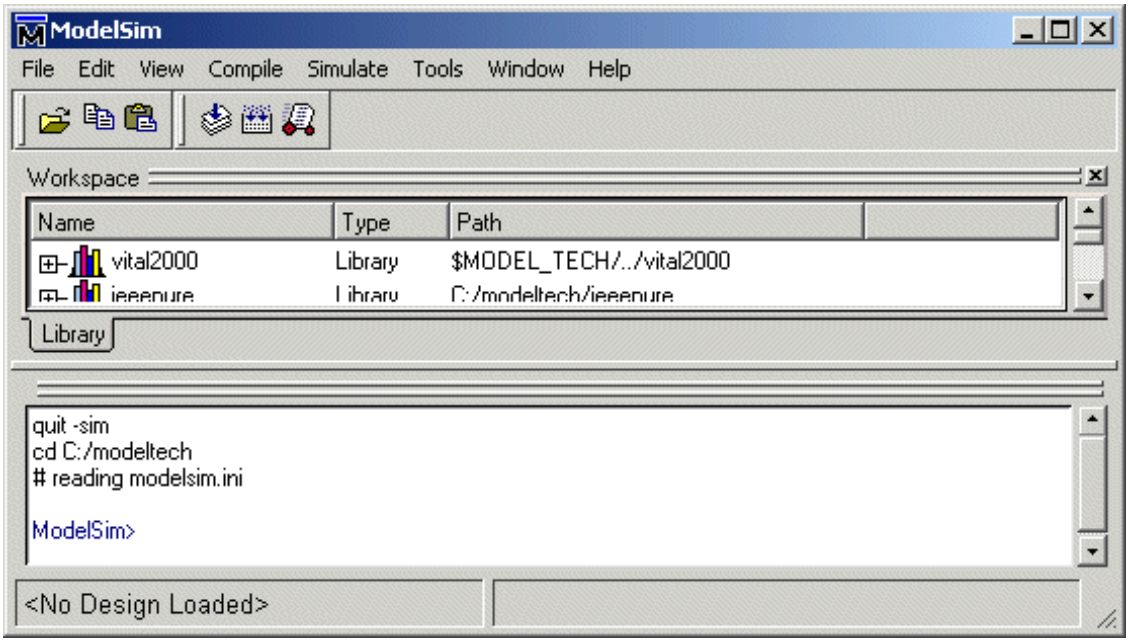
- "Finding items by name in the List window" (UM-177),
- "Finding HDL items in the Signals window" (UM-188), and
- "Finding items by name or value in the Wave window" (UM-225).

Main window

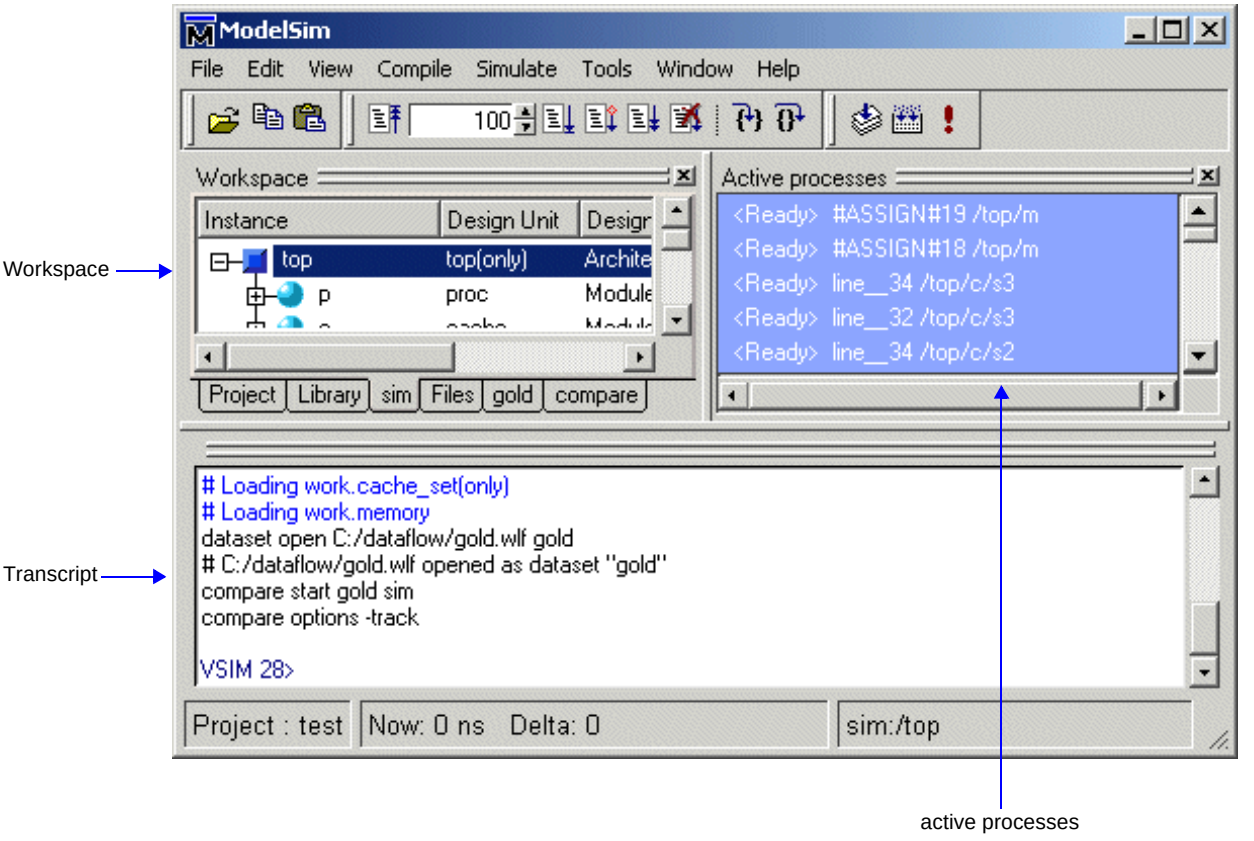
The Main window is pictured below as it appears when ModelSim is first invoked. Note that your operating system graphic interface provides the window-management frame only; ModelSim handles all internal-window features including menus, buttons, and scroll bars.



You can customize the Main window layout–click and drag on the bars noted in the graphic above to change the position of the panes and toolbars. You can also change the relative size of each pane by dragging on its border. The graphic below shows a customized layout.



The graphic below shows the Main window as it might appear when you have a project and a design loaded.



The menu bar at the top of the window provides access to a wide variety of simulation commands and ModelSim preferences. The toolbar provides buttons for quick access to the many common commands. The status bar at the bottom of the window gives you information about the data in the active ModelSim window. The panes display different parts of your design or different features of ModelSim. The panes, menu bar, toolbar, and status bar are described in detail below.

Workspace

The Workspace is available in ModelSim versions 5.5 and later. It provides convenient access to projects, libraries, design files, compiled design units, simulation/dataset structures, and Waveform Comparison objects. It can be hidden or displayed by selecting **View** > **Workspace** (Main window).

The Workspace can display five types of tabs as shown in the graphic above.

- **Project tab**
  Shows all files that are included in the open project. See *Chapter 2 - Projects* for details.
- **Library tab**
  Shows design libraries and compiled design units. See "Managing library contents" (UM-41) for details.

- **Structure tabs**
  Shows a hierarchical view of the active simulation and any open datasets. This is the same data that is displayed in the "Structure window" (UM-199). There is one tab for the current simulation and one tab for each open dataset. See "Viewing dataset structure" (UM-120) for details.

Transcript

The Transcript portion of the Main window maintains a running history of commands that are invoked and messages that occur as you work with ModelSim. When a simulation is running, the Transcript displays a VSIM prompt, allowing you to enter command-line commands from within the graphic interface.

You can scroll backward and forward through the current work history by using the vertical scrollbar. You can also use arrow keys to recall previous commands, or copy and paste using the mouse within the window (see "Mouse and keyboard shortcuts" (UM-147) for details).

Saving the Main window transcript file

Variable settings determine the filename used for saving the Main window transcript. If either **PrefMain(file)** in the *modelsim.tcl* file or **TranscriptFile** in the *modelsim.ini* file is set, then the transcript output is logged to the specified file. By default the **TranscriptFile** variable in *modelsim.ini* is set to *transcript*. If either variable is set, the transcript contents are always saved and no explicit saving is necessary.

If you would like to save an additional copy of the transcript with a different filename, you can use the **File > Transcript > Save Transcript As**, or **File > Transcript > Save Transcript** menu items. The initial save must be made with the **Save Transcript As** selection, which stores the filename in the Tcl variable **PrefMain(saveFile)**. Subsequent saves can be made with the **Save Transcript** selection. Since no automatic saves are performed for this file, it is written only when you invoke a **Save** command. The file is written to the specified directory and records the contents of the transcript at the time of the save.

Using the saved transcript as a macro (DO file)

Saved transcript files can be used as macros (DO files). See the **do** command (CR-68) for more information.

Active processes

This pane displays all processes that are scheduled to run during the current simulation cycle. You can hide or display this pane by selecting **View > Active Process** (Main window). This same data can be displayed in the "Process window" (UM-181).

The Main window menu bar

The menu bar at the top of the Main window lets you access many ModelSim commands and features. The menus are listed below with brief descriptions of each command's use.

File menu

New	provides these options: Folder – create a new folder in the current directory Source – create a VHDL, Verilog, or Other source file Project – create a new project Library – create a new design library and mapping; see "Creating a library" (UM-40)
Open	provides these options: File – open the selected hdl file Project – open the selected *.mpf* project file Dataset – open the specified WLF file and assign it the specified dataset name Exclusion File – open "Exclusion filter files" (UM-298) for Code Coverage
Close	provides these options: Project – close the currently open project file Dataset – close the specified dataset
Import	provides this option: Library – import FPGA libraries; see "Importing FPGA libraries" (UM-48)
Save	provides these options: sim dataset – save data from the current simulation Exclusion File – save "Exclusion filter files" (UM-298) for Code Coverage
Delete	provides this option: Project – delete the selected *.mpf* project file
Change Directory	change to a different working directory
Transcript	provides these options: Save Transcript – save the Main window transcript to the file indicated with a "Save Transcript As" selection (this selection is not initially available because the transcript is written to the *transcript* file by default), see "Saving the Main window transcript file" (UM-139) Save Transcript As – save the Main window transcript to a file Clear Transcript – clear the Main window transcript display Print – print the contents of the Transcript window
Add to Project	provides these options: File – add files to the open Project; see "Step 2 — Adding items to the project" (UM-21) Simulation Configuration – add an object representing a design unit(s) and its associated simulation options; see "Creating a Simulation Configuration" (UM-30) Folder – add an organization folder to the current project; see "Organizing projects with folders" (UM-32)

Recent Directories Recent Projects	display a list of the most recent working directories or projects, respectively
Quit	quit ModelSim

Edit menu

Copy	copy the selected text
Paste	paste the previously cut or copied text
Select All	select all text in the Main window transcript
Unselect All	deselect all text in the Main window transcript
Find	search the transcript forward or backward for the specified text string

View menu

All Windows	open all ModelSim windows
Dataflow	open and/or view the Dataflow window (UM-149)
List	open and/or view the List window (UM-168)
Process	open and/or view the Process window (UM-181)
Signals	open and/or view the Signals window (UM-183)
Source	open and/or view the Source window (UM-191)
Structure	open and/or view the Structure window (UM-199)
Variables	open and/or view the Variables window (UM-203)
Wave	open and/or view the Wave window (UM-206)
Datasets	open the Dataset Browser to open, close, rename, or activate a dataset
Coverage	provides these options: Current Exclusions – hide or show the Exclusions pane Missed Coverage – hide or show the Missed Coverage pane Instance Coverage – hide or show the Instance Coverage pane
Active Process	hide or show the Active processes (UM-139) pane
Workspace	hide or show the Workspace (UM-138)
Encoding	select from alphabetical list of encoding names that enable proper display of character representations used by various operating systems or file systems, such as Unicode, ASCII, or Shift-JIS.
Properties	show information about the item selected in the workspace

Compile menu

Compile	compile HDL source files; not enabled if you have a project open
Compile Options	set both VHDL and Verilog compile options; disabled if you have a project open
Compile All	compile all files in the open project; see "Step 3 — Compiling the files" (UM-24) for details
Compile Selected	compile the files selected in the project tab; disabled if you don't have a project open
Compile Order	set the compile order of the files in the open project; see "Changing compile order" (UM-28) for details
Compile Report	report on the compilation history of the selected file(s) in the project
Compile Summary	report on the compilation history of all files in the project

Simulate menu

Simulate	load the selected design unit; see Simulating with the graphic interface (UM-245)
Simulation Options	set various simulation options;
Run	provides seven options: Run <default> – run simulation for one default run length; change the run length with **Simulate > Simulation Options**, or use the Run Length text box on the toolbar Run -All – run simulation until you stop itContinue – continue the simulationRun -Next – run to the next event time Step – single-step the simulatorStep -Over – execute without single-stepping through a subprogram call Restart – reload the design elements and reset the simulation time to zero; only design elements that have changed are reloaded; you specify whether to maintain the following after restart–List and Wave window environment, breakpoints, logged signals, and virtual definitions; see also the **restart** command (CR-111)
Break	stop the current simulation run
End Simulation	quit the current simulation run

Tools menu

Breakpoints	open the Breakpoints dialog box; see "Setting file-line breakpoints" (UM-197) for details
Options (all options are set for the current session only)	provides these options: Transcript File – set a transcript file to save for this session only Command History – set a file for saving command history only, no comments Save File – set filename for Save Transcript, and Save Transcript As Saved Lines – limit the number of lines saved in the transcript (default is 5000) Line Prefix – specify the comment prefix for the transcript Update Rate – specify the update frequency for the Main status bar ModelSim Prompt – change the title of the ModelSim prompt VSIM Prompt – change the title of the VSIM prompt Paused Prompt – change the title of the Paused prompt HTML Viewer – specify the path to your browser; used for displaying online help
Edit Preferences	set various preference variables; see http://www.model.com/resources/pref_variables/frameset.htm
Save Preferences	save current ModelSim settings to a Tcl preference file; http://www.model.com/resources/pref_variables/frameset.htm

Window menu

Initial Layout	restore all windows to the size and placement of the initial full-screen layout
Cascade	cascade all open windows
Tile Horizontally	tile all open windows horizontally
Tile Vertically	tile all open windows vertically
Layout Style[a]	provides these options: Default - restore the windows to version 5.5 layout Millennium - restore the windows to version 5.6 layout Classic - restore the windows to pre-5.5 layout Cascade - cascade all open windows Horizontal - tile all open windows horizontally Vertical - tile all open windows vertically
Icon Children	icon all but the Main window
Icon All	icon all windows
Deicon All	deicon all windows
<window_name>	list of up to nine open windows including one for each file opened in the Source window; use the Windows menu item to see a complete list
Windows	open dialog with complete list of open windows

a.You can specify a Layout Style to become the default for ModelSim. After choosing the Layout Style you want, select **Tools > Save Preferences** and the layout style will be saved to the PrefMain(layoutStyle) preference variable.

Help menu

About ModelSim	display ModelSim application information (e.g., software version)
Release Notes	view current release notes with the ModelSim **notepad** (CR-95)
Welcome Menu	open the Welcome screen
Documentation	open and read ModelSim documentation in PDF or HTML format; PDF files can be read with a free Adobe Acrobat reader available on the ModelSim installation CD or from www.adobe.com
Tcl Help	open the Tcl command reference (man pages) in Windows help format
Tcl Man Pages	open the Tcl /Tk 8.3 manual in HTML format
Technotes	select a technical note to view from the drop-down list

The Main window toolbar

Buttons on the Main window toolbar give you quick access to these ModelSim commands and functions.

Main window toolbar buttons

Button	Menu equivalent	Command equivalents
Open open the Open File dialog	File > Open > File	
Copy copy the selected text within the Main window transcript	Edit > Copy	see: "Mouse and keyboard shortcuts" (UM-147)
Paste paste the copied text to the cursor location	Edit > Paste	see: "Mouse and keyboard shortcuts" (UM-147)
Compile open the Compile HDL Source Files dialog box to select files for compilation	Compile > Compile	**vcom** <arguments>, or **vlog** <arguments> see: **vcom** (CR-145) or **vlog** (CR-181)
Compile All compile all files in the open project	Compile > Compile	**vcom** <arguments>, or **vlog** <arguments> see: **vcom** (CR-145) or **vlog** (CR-181)
Simulate load the selected design unit or simulation configuration object	Simulate > Simulate	**vsim** <arguments> see: **vsim** (CR-189)
Restart reload the design elements and reset the simulation time to zero, with the option of using current formatting, breakpoints, and WLF file	Simulate > Run > Restart	**restart** <arguments> see: **restart** (CR-111)
Run Length specify the run length for the current simulation	Simulate > Simulation Options	**run** <specific run length> see: **run** (CR-114)

Main window toolbar buttons			
Button		**Menu equivalent**	**Command equivalents**
	Run run the current simulation for the specified run length	Simulate > Run > Run <default_run_length>	**run** (no arguments) see: **run** (CR-114)
	Continue Run continue the current simulation run until the end of the specified run length or until it hits a breakpoint or specified break event	Simulate > Run > Continue	**run -continue** see: **run** (CR-114)
	Run -All run the current simulation forever, or until it hits a breakpoint or specified break event	Simulate > Run > Run -All	**run -all** see: **run** (CR-114), see "Assertions tab" (UM-255)
	Break stop the current simulation run	Simulate > Break	none
	Step step the current simulation to the next HDL statement	Simulate > Run > Step	**step** see: **step** (CR-122)
	Step Over HDL statements are executed but treated as simple statements instead of entered and traced line by line	Simulate > Run > Step -Over	**step -over** see: **step** (CR-122)

The Main window status bar

Fields at the bottom of the Main window provide the following information about the current simulation:

Field	Description
Project	name of the current project
Now	the current simulation time, using the default resolution units (see "Simulating with the graphic interface" (UM-245)), or a larger time unit if one can be used without a fractional remainder
Delta	the current simulation iteration number
environment	name of the current context (item selected in the Structure window (UM-199))

Mouse and keyboard shortcuts

The following mouse actions and special keystrokes can be used to edit commands in the entry region of the Main window. They can also be used in editing the file displayed in the Source window and all Notepad windows (enter the **notepad** command within ModelSim to open the Notepad editor).

Keystrokes	Result
< left | right - arrow >	move the cursor left | right one character
< up | down - arrow >	scroll through command history (in Source window, move cursor one line up | down)
< control > < left | right - arrow >	move cursor left | right one word
< shift > < left | right | up | down - arrow >	extend selection of text
< control > < shift > < left | right - arrow >	extend selection of text by word
< up | down - arrow >	scroll through command history (in Source window, moves cursor one line up | down)
< control > < up | down >	move cursor up | down one paragraph
< alt >	activate or inactivate menu bar mode
< alt > < F4 >	close active window
< backspace >	delete character to the left
< home >	move cursor to the beginning of the line

Keystrokes	Result
< end >	move cursor to the end of the line
< control > < home >	move cursor to the beginning of the text
< control > < end >	move cursor to the end of the text
< esc >	cancel
< control - a >	select the entire content of the widget
< control - c >	copy the selection
< control - f >	find
< F3 >	find next
< control - k >	delete from the cursor to the end of the line
< control - s >	save
< control - t >	reverse the order of the two characters to the right of the cursor
< control - u >	delete line
< control - v >	paste from the clipboard
< control - x >	cut the selection
< F8 >	search for the most recent command that matches the characters typed
< F9 >	run simulation
< F10 >	continue simulation
< F11 >	single-step
< F12 >	step-over

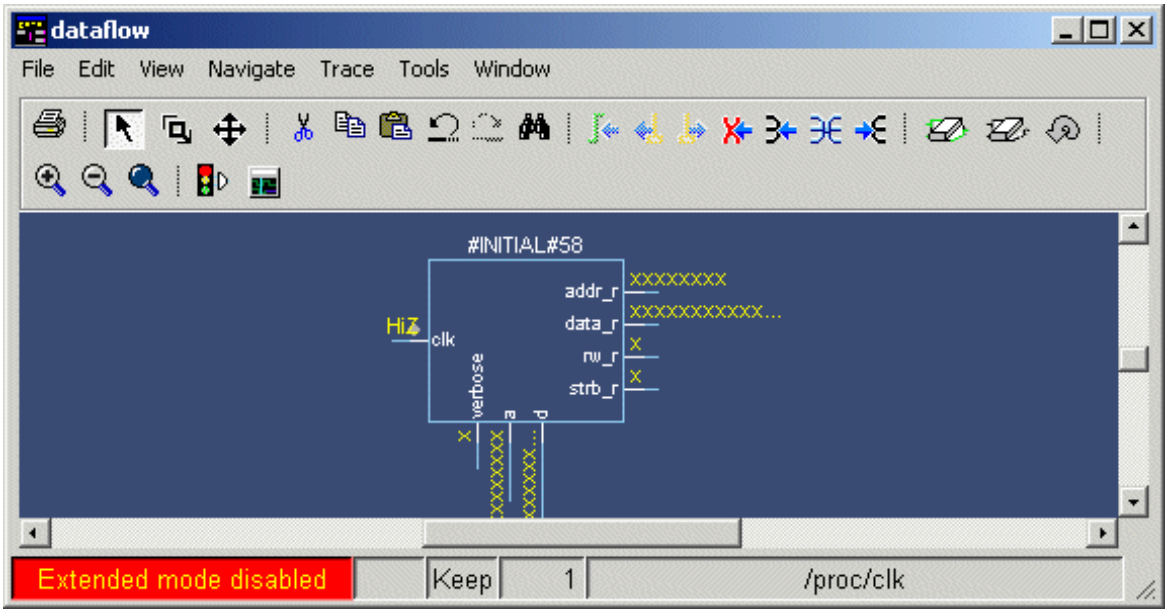
The Main window allows insertions or pastes only after the prompt; therefore, you don’t need to set the cursor when copying strings to the command line.

Dataflow window

The Dataflow window allows you to explore the "physical" connectivity of your design. The window displays processes and signals, nets, and registers.

Note: OEM versions of ModelSim have limited Dataflow functionality. Many of the features described below will operate differently. The window will show only one process and its attached signals or one signal and its attached processes, as displayed in the graphic below.

Adding items to the window



You can use any of the following methods to add items to the Dataflow window:

- drag and drop items from other windows
- use the Navigate menu options in the Dataflow window
- use the **add dataflow** command (CR-31)
- double-click any waveform in the Wave window display

The **Navigate** menu offers four commands that will add items to the window. The commands include:

View region — clear the window and display all signals from the current region

Add region — display all signals from the current region without first clearing window

View all nets — clear the window and display all signals from the entire design

Add ports — add port symbols to the port signals in the current region

When you view regions or entire nets, the window initially displays only the drivers of the added items in order to reduce clutter. You can easily view readers by selecting an item and invoking **Navigate > Expand net to readers**.

A small circle above an input signal on a block denotes a trigger signal that is on the process' sensitivity list.

Links to other windows

The Dataflow window has links to other windows as described below:

Window	Link
Main window (UM-137)	select a signal or process in the Dataflow window, and the Structure pane updates if that item is in a different design unit
Process window (UM-181)	select a process in either window, and that process is highlighted in the other
Signals window (UM-183)	select a signal in either window, and that signal is highlighted in the other
Wave window (UM-206)	• trace through the design in the Dataflow window, and the associated signals are added to the Wave window • move a cursor in the Wave window, and the values update in the Dataflow window
Source window (UM-191)	select an item in the Dataflow window, and the Source window updates if that item is in a different source file

Dataflow window menu bar

The following menu commands are available from the Dataflow window menu bar. Many of the commands are also available from the context menu (click right or 3rd mouse button).

File menu

Print	print the current view of the Dataflow window
Print Postscript	print/save the current view of the Dataflow window to a postscript device/file
Page setup	configure page formatting for printing
Close	close the Dataflow window; note that this erases whatever is currently displayed in the window

Edit menu

Undo	undo the last action
Redo	redo the last undone action
Cut	cut the selected object(s)
Copy	copy the selected object(s)
Paste	paste the previously cut or copied object(s) into the display
Erase selected	clear selected object from window
Select all	select all objects in the window
Unselect all	deselect all currently selected objects
Erase highlight	remove green highlighting from interconnect lines
Erase all	clear all objects from window
Regenerate	clear and redraw the display using an optimal layout
Find	search for an instance or signal
Find Next	search for next occurrence of instance or signal

View menu

Show Wave	open the embedded wave viewer pane
Select	set left mouse button to select mode and middle mouse button to zoom mode
Zoom	set left mouse button to zoom mode and middle mouse button to pan mode
Pan	set left mouse button to pan mode and middle mouse button to zoom mode
Default	set mouse to default mode

Navigate menu

Expand net to drivers	display driver(s) of the selected signal, net, or register
Expand net to readers	display reader(s) of the selected signal, net, or register
Expand net	display driver(s) and reader(s) of the selected signal, net, or register

Hide selected	remove the selected component and all other components from the same region and replace them with a single component representing that region
Show selected	expand the selected component to show all underlying components
View region	clear the window and display all signals from the current region
Add region	display all signals from the current region without first clearing the window
View all nets	clear the window and display all signals from the entire design
Add ports	add port symbols to the port signals in the current region

Trace menu

TraceX™	step back to the last driver of an unknown (X) value
ChaseX™	jump to the source of an unknown (X) value
TraceX Delay	step back in time to the last driver of an unknown (X) value
ChaseX Delay	jump back in time to the point where the output value transitions to X
Trace next event	move the next event cursor to the next input event driving the selected output
Trace event set	jump to the source of the selected input event
Trace event reset	return the next event cursor to the selected output

Tools menu

Load built-in symbol map	load a .bsm file for mapping symbol instances; see "Symbol mapping" (UM-165)
Load symlib library	load a user-defined symbol library
Create symlib index	create an index for a user-defined symbol library
Options	configure Dataflow window preferences

Window menu

The Window menu is identical in all windows. See "Window menu" (UM-144) for a description of the commands.

The Dataflow window toolbar

The buttons on the Dataflow window toolbar are described below.

Button	Menu equivalent
Print print the current view of the Dataflow window	File > Print
Select mode set left mouse button to select mode and middle mouse button to zoom mode	View > Select
Zoom mode set left mouse button to zoom mode and middle mouse button to pan mode	View > Zoom
Pan mode set left mouse button to pan mode and middle mouse button to zoom mode	View > Pan
Cut cut the selected object(s)	Edit > Cut
Copy copy the selected object(s)	Edit > Copy
Paste paste the previously cut or copied object(s)	Edit > Paste
Undo undo the last action	Edit > Undo
Redo redo the last undone action	Edit > Redo
Find search for an instance or signal	Edit > Find

Button	Menu equivalent
Trace input net to event move the next event cursor to the next input event driving the selected output	Trace > Trace next event
Trace Set jump to source of selected input event	Trace > Trace event set
Trace Reset return the next event cursor to the selected output	Trace > Trace event reset
Trace net to driver of X step back to the last driver of an unknown value	Trace > TraceX
Expand net to all drivers display driver(s) of the selected signal, net, or register	Navigate > Expand net to drivers
Expand net to all drivers and readers display driver(s) and reader(s) of the selected signal, net, or register	Navigate > Expand net
Expand net to all readers display reader(s) of the selected signal, net, or register	Navigate > Expand net to readers
Erase highlight clear the green highlighting which identifies the path you've traversed through the design	Edit > Erase highlight
Erase all clear the window	Edit > Erase all
Regenerate clear and redraw the display using an optimal layout	Edit > Regenerate

Button	Menu equivalent
Zoom In zoom in by a factor of two from current view	none
Zoom Out zoom out by a factor of two from current view	none
Zoom Full zoom out to show all components in window	none
Stop Drawing halt any drawing currently happening in the window	none
Show Wave display the embedded wave viewer pane	View > Show Wave

Exploring the connectivity of your design

A primary use of the Dataflow window is exploring the "physical" connectivity of your design. One way of doing this is by expanding the view from process to process. This allows you to see the drivers/receivers of a particular signal, net, or register.

You can expand the view of your design using menu commands or your mouse. To expand with the mouse, simply double click a signal, register, or process. Depending on the specific item you click, the view will expand to show the driving process and interconnect, the reading process and interconnect, or both.

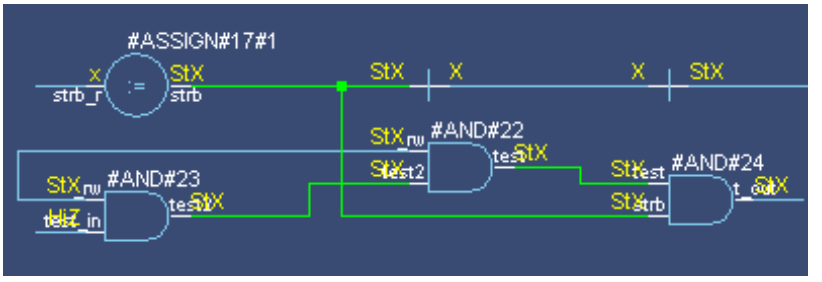
Alternatively, you can select a signal, register, or net, and use one of the toolbar buttons or menu commands described below:

	Expand net to all drivers display driver(s) of the selected signal, net, or register	Navigate > Expand net to drivers
	Expand net to all drivers and readers display driver(s) and reader(s) of the selected signal, net, or register	Navigate > Expand net
	Expand net to all readers display reader(s) of the selected signal, net, or register	Navigate > Expand net to readers

As you expand the view, note that the "layout" of the design may adjust to best show the connectivity. For example, the location of an input signal may shift from the bottom to the top of a process.

Tracking your path through the design

You can quickly traverse through many components in your design. To help mark your path, the items that you have expanded are highlighted in green.



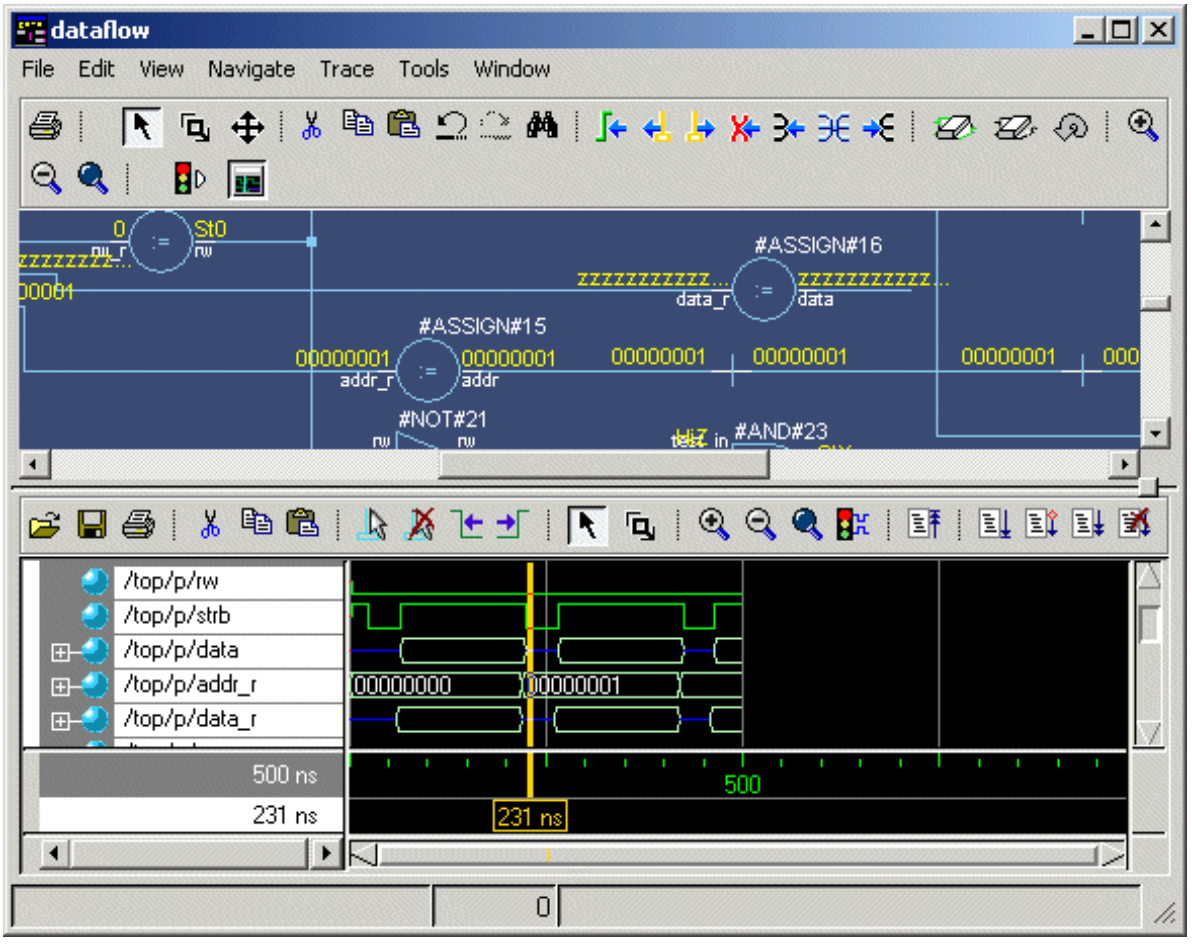
You can clear this highlighting using the **Edit > Erase highlight** command.

The embedded wave viewer

Another way of exploring your design is to use the Dataflow window's embedded wave viewer. This viewer closely resembles, in appearance and operation, the stand-alone Wave window (see "Wave window" (UM-206) for more information).

The wave viewer is opened using the **View > Show Wave** command.

One common scenario is to place signals in the wave viewer and the Dataflow panes, run the design for some amount of time, and then use time cursors to investigate value changes. In other words, as you place and move cursors in the wave viewer pane (see "Using time cursors in the Wave window" (UM-226) for details), the signal values update in the Dataflow pane.



Another scenario is to select a process in the Dataflow pane, which automatically adds to the wave viewer pane all signals attached to the process.

See "Tracing events (causality)" (UM-159) for another example of using the embedded wave viewer.

Zooming and panning

The Dataflow window offers several tools for zooming and panning the display.

Zooming with toolbar buttons

These zoom buttons are available on the toolbar:

Zoom In zoom in by a factor of two from the current view	**Zoom Out** zoom out by a factor of two from current view
Zoom Full zoom out to view the entire schematic	

Zooming with the mouse

To zoom with the mouse, you can either use the middle mouse button or enter Zoom Mode by selecting **View > Zoom** and then use the left mouse button.

Four zoom options are possible by clicking and dragging in different directions:

- Down-Right: Zoom Area (In)
- Up-Right: Zoom Out (zoom amount is displayed at the mouse cursor)
- Down-Left: Zoom Selected
- Up-Left: Zoom Full

The zoom amount is displayed at the mouse cursor. A zoom operation must be more than 10 pixels to activate.

Panning with the mouse

To pan with the mouse you must enter Pan Mode by selecting **View > Pan**.

Now click and drag with the left mouse button to pan the design.

Tracing events (causality)

One of the most useful features of the Dataflow window is tracing an event to see the cause of an unexpected output. This feature uses the Dataflow window's embedded wave viewer (see "The embedded wave viewer" (UM-157) for more details).

In short you identify an output of interest in the Dataflow pane and then use time cursors in the wave viewer pane to identify events that contribute to the output.

The process for tracing events is as follows:

1. Log all signals before starting the simulation (add log -r /*).
2. After running a simulation for some period of time, open the Dataflow window and the wave viewer pane.
3. Add a process or signal of interest into the Dataflow window (if adding a signal, find its driving process). Select the process and all signals attached to the selected process will appear in the wave viewer pane.
4. Place a time cursor on an edge of interest; the edge should be on a signal that is an output of the process.
5. Select **Trace > Trace next event**.

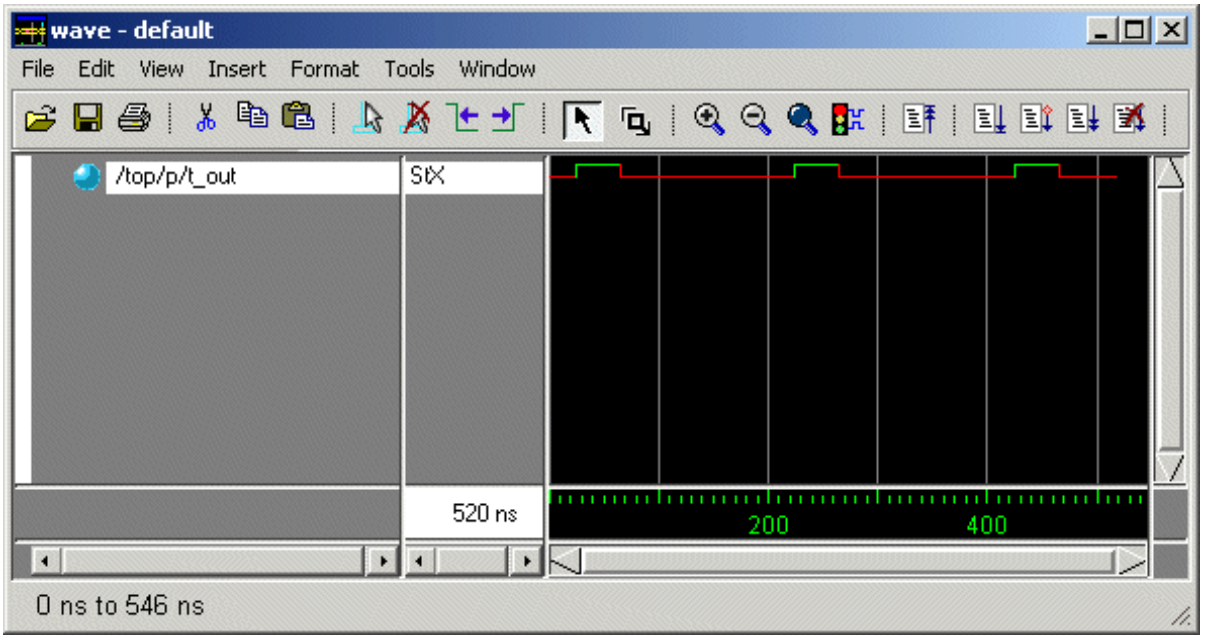
   A second cursor is added at the most recent input event.
6. Keep selecting **Trace > Trace next event** until you've reached an input event of interest. Note that the signals with the events are selected in the wave pane.
7. Now select **Trace > Trace set**.

   The Dataflow display "jumps" to the source of the selected input event(s). The operation follows all signals selected in the wave viewer pane. You can change which signals are followed by changing the selection.
8. To continue tracing, go back to step 5 and repeat.

If you want to start over at the originally selected output, select **Trace > Trace reset**.

Tracing the source of an unknown (X)

Another useful debugging option is locating the source of an unknown (X). Unknown values are most clearly seen in the Wave window—the waveform displays in red when a value is unknown.



The procedure for tracing an unknown is as follows:

1. Load your design.
2. Log all signals in the design or any signals that may possibly contribute to the unknown value (**log -r /*** will log all signals in the design).
3. Add signals to the Wave window or wave viewer pane, and run your design the desired length of time.
4. Put a cursor on the time at which the signal value is unknown.
5. Add the signal of interest to the Dataflow window, making sure the signal is selected.
6. Select **Trace > TraceX**, **Trace > TraceX Delay, Trace > ChaseX**, or **Trace > ChaseX Delay**.

These commands behave as follows:

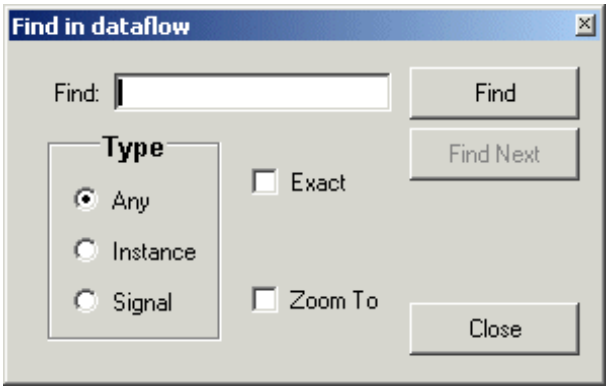
TraceX / TraceX Delay— Step back to the last driver of an X value. **TraceX Delay** works similarly but it steps back in time to the last driver of an X value. **TraceX** should be used for RTL designs; **TraceX Delay** should be used for gate-level netlists with backannotated delays.

Trace > ChaseX / ChaseX Delay — "Jumps" through a design from output to input, following X values. **ChaseX Delay** acts the same as **ChaseX** but also moves backwards in

time to the point where the output value transitions to X.**ChaseX** should be used for RTL designs; **ChaseX Delay** should be used for gate-level netlists with backannotated delays.

Finding items by name in the Dataflow window

Select **Edit > Find** to search for signal, net, or register names or an instance of a component.



Enter an item name and specify whether it is an instance of a process (Instance); a signal, net, or register (Signal); or either (Any).

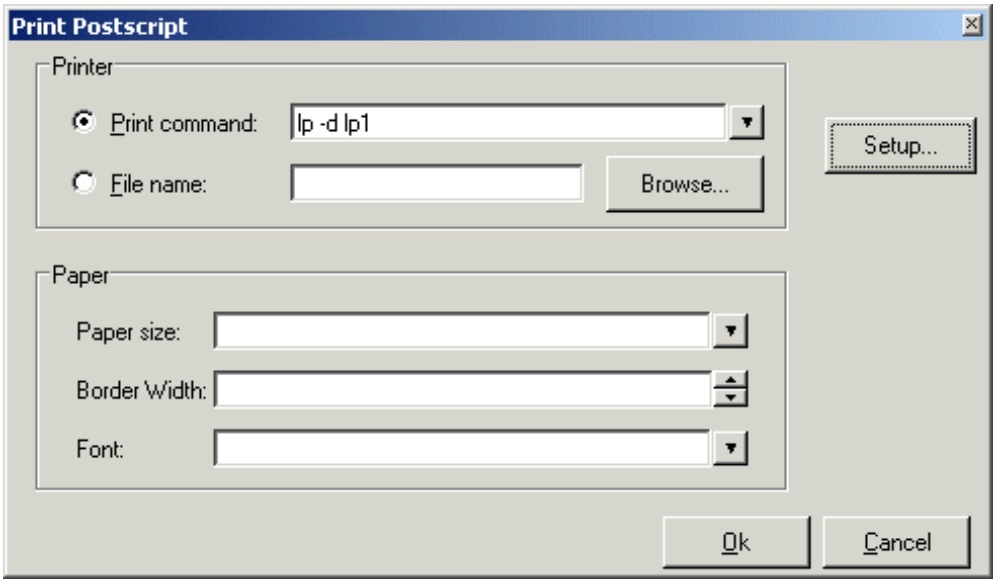
Specify **Exact** if you only want to find items that match your search exactly. For example, searching for "clk" without **Exact** will find */top/clk* and *clk1*.

If you want to zoom in on the located item, select Zoom To. You can continue searching using the Find Next button.

Saving the display

Saving a .eps file

Select **File** > **Print Postscript** to save the waveform as a .eps file.



The **Print Postscript** dialog box includes these options:

Printer

- **File name**
  Enter a filename for the encapsulated Postscript (.eps) file to create; or browse to a previously created .eps file and use that filename.

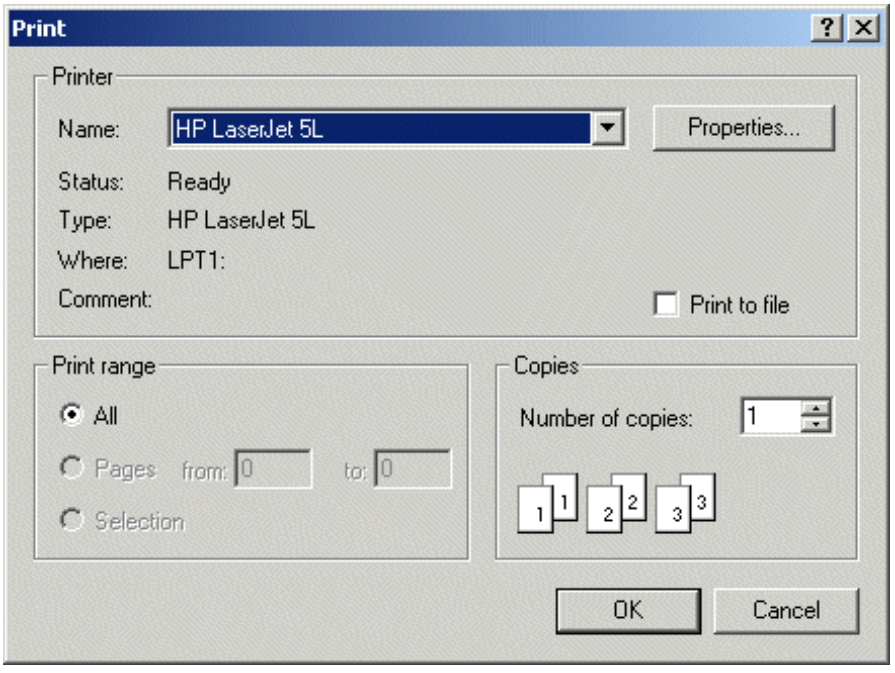
Paper

Setup button

See "Printer Page Setup" (UM-236).

Printing on Windows platforms

Select **File** > **Print** to print the Dataflow display or to save the display to a file.



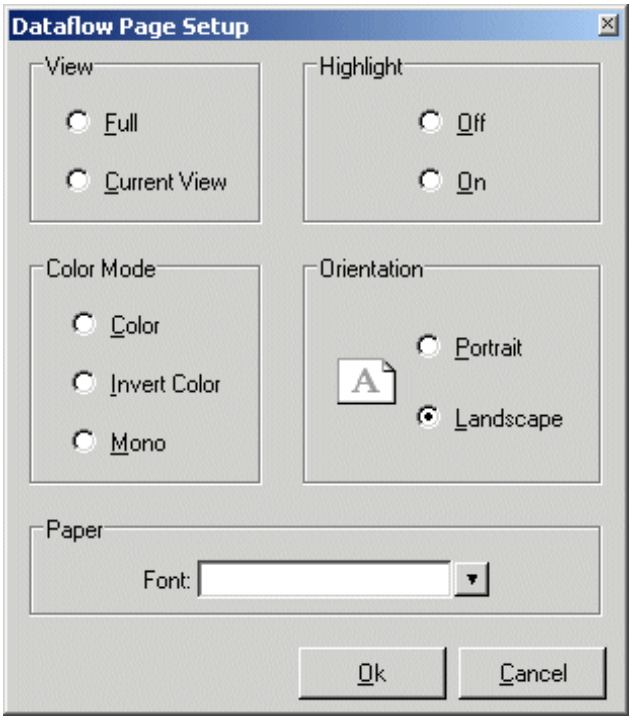
The **Print** dialog box includes these options:

Printer

- **Name**
  Choose the printer from the drop-down menu. Set printer properties with the *Properties* button.
- **Status**
  Indicates the availability of the selected printer.
- **Type**
  Printer driver name for the selected printer. The driver determines what type of file is output if "Print to file" is selected.
- **Where**
  The printer port for the selected printer.
- **Comment**
  The printer comment from the printer properties dialog box.
- **Print to file**
  Make this selection to print the display to a file instead of a printer. The printer driver determines what type of file is created. Postscript printers create a Postscript (.ps) file, non-Postscript printers create a .prn or printer control language file. To create an encapsulated Postscript file (.eps) use the **File** > **Print Postscript** menu selection.

Configuring page setup

Clicking the Setup button in the Print Postscript or Print dialog box allows you to define the following options (this is the same dialog that opens via **File > Page setup**).



The **Dataflow Page Setup** dialog box includes these options:

- **View**
  Specifies **Full** (everything in the window) or **Current View** (only that which is visible).
- **Highlight**
  Specifies that highlighting (see "Tracking your path through the design" (UM-156)) is **On** or **Off**.
- **Color Mode**
  Specifies **Color** (256 colors), **Invert Color** (gray-scale) or **Mono** (monochrome) color mode.
- **Orientation**
  Specifies **Landscape** (horizontal) or **Portrait** (vertical) orientation.
- **Paper**
  Specifies the font to use for printing.

Symbol mapping

The Dataflow window has built-in mappings for all Verilog primitive gates (i.e., AND, OR, etc.). For components other than Verilog primitives, you can define a mapping between processes and built-in symbols. This is done through a file containing name pairs, one per line, where the first name is the concatenation of the design unit and process names, (DUname.Processname), and the second name is the name of a built-in symbol. For example:

```
xorg(only).p1 XOR
org(only).p1 OR
andg(only).p1 AND
```

Entities and modules are mapped the same way:

```
AND1 AND
AND2 AND  # A 2-input and gate
AND3 AND
AND4 AND
AND5 AND
AND6 AND
xnor(test) XNOR
```

Note that for primitive gate symbols, pin mapping is automatic.

The Dataflow window looks in the current working directory and inside each library referenced by the design for the file *dataflow.bsm* (.bsm stands for "Built-in Symbol Map). It will read all files found.

User-defined symbols

You can also define your own symbols using an ASCII symbol library file format for defining symbol shapes. This capability is delivered via Concept Engineering's Nlview™ widget Symlib format. For more specific details on this widget, see www.model.com/products/documentation/nlviewSymlib.html.

The Dataflow window will search the current working directory, and inside each library referenced by the design, for the file *dataflow.sym*. Any and all files found will be given to the Nlview widget to use for symbol lookups. Again, as with the built-in symbols, the DU name and optional process name is used for the symbol lookup. Here's an example of a symbol for a full adder:

```
symbol adder(structural) * DEF \
    port a in -loc -12 -15 0 -15 \
    pinattrdsp @name -cl 2 -15 8 \
    port b in -loc -12 15 0 15 \
    pinattrdsp @name -cl 2 15 8 \
    port cin in -loc 20 -40 20 -28 \
    pinattrdsp @name -uc 19 -26 8 \
    port cout out -loc 20 40 20 28 \
pinattrdsp @name -lc 19 26 8 \
    port sum out -loc 63 0 51 0 \
    pinattrdsp @name -cr 49 0 8 \
    path 10 0 0 7 \
    path 0 7 0 35 \
    path 0 35 51 17 \
    path 51 17 51 -17 \
    path 51 -17 0 -35 \
    path 0 -35 0 -7 \
    path 0 -7 10 0
```

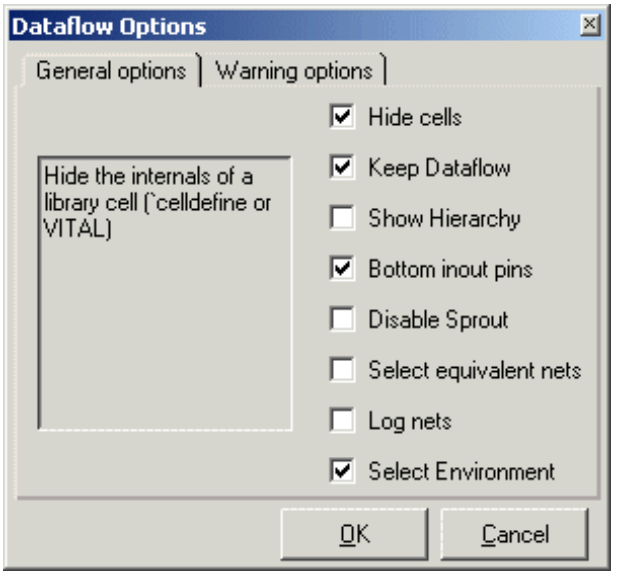
Port mapping is done by name for these symbols, so the port names in the symbol definition must match the port names of the Entity|Module|Process (in the case of the process, it's the signal names that the process reads/writes).

Important: When you create or modify a symlib file, you must generate a file index. This index is how the Nlview widget finds and extracts symbols from the file. To generate the index, select **Tools > Create symlib index** (Dataflow window) and specify the symlib file. The file will be rewritten with a correct, up-to-date index.

Configuring window options

You can configure several options that determine how the Dataflow window behaves. The settings affect only the current session.

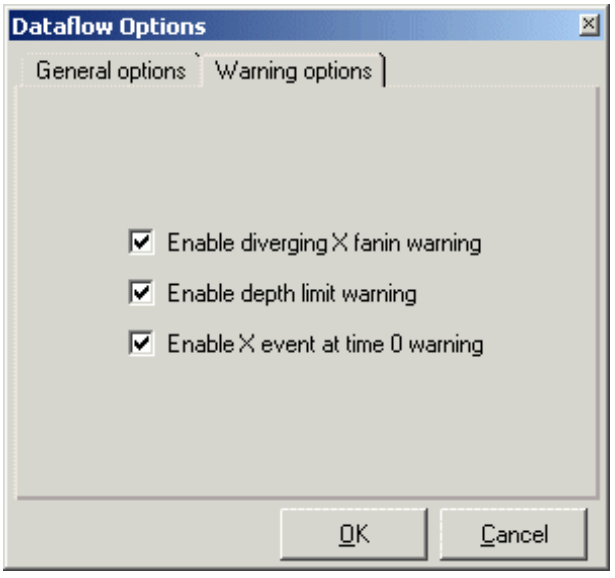
Select **Tools > Options** to open the Dataflow Options dialog box.



The **General options** tab includes these options:

- **Hide Cells**
  By default the Dataflow window automatically hides instances that have either 'celldefine, VITAL_LEVEL0, or VITAL_LEVEL1 attributes. Unchecking this disables automatic cell hiding.
- **Keep Dataflow**
  Keeps previous contents when adding new signals or processes to the window.
- **Show Hierarchy**
  Displays connectivity using hierarchical references. Note that selecting this will erase the current contents of the window.
- **Bottom inout pins**
  Places inout pins on the bottom of components rather than on the right with output pins.

- **Disable Sprout**
  Displays only the selected signal or process with its immediate fanin/fanout. Configures window to behave like the Dataflow window of versions prior to 5.6.
- **Select equivalent nets**
  If the item you select traverses hierarchy, then ModelSim selects all connected items across the hierarchy.
- **Log nets**
  Logs signals when they are added to the window.
- **Select environment**
  Updates the Structure, Signals, and Source windows to reflect the net selected in the Dataflow window.

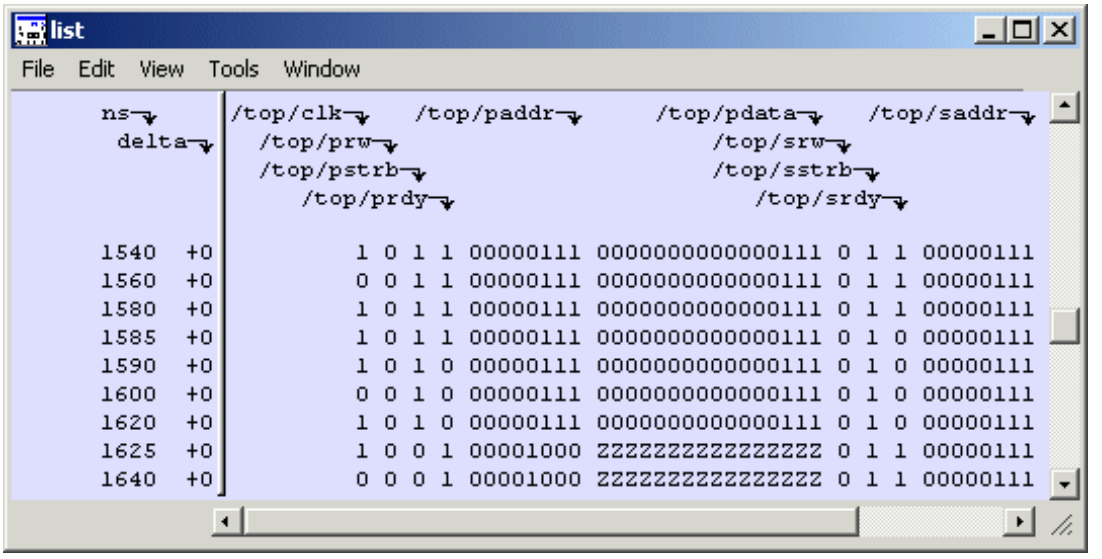


The **Warning options** tab includes these options:

- **Enable diverging X fanin warning**
  Enables the warning message, "ChaseX: diverging X fanin. Reduce the selection list and try again."
- **Enable depth limit warning**
  Enables the warning message, "ChaseX: Stop because depth limit reached! Possible loop?"
- **Enable X event at time 0 warning**
  Enables the warning message, ""Driving X event at time 0."

List window

The List window displays the results of your simulation run in tabular format. The window is divided into two adjustable panes, which allow you to scroll horizontally through the listing on the right, while keeping time and delta visible on the left.



HDL items you can view

One entry is created for each of the following items within the design:

- VHDL
  signals and process and shared variables
- Verilog
  nets, registers, and variables
- Virtuals
  Virtual signals and functions

Note: Constants, generics, and parameters are not viewable in the List or Wave windows.

Adding HDL items to the List window

Before adding items to the List window you may want to set the window display properties (see "Setting List window display properties" (UM-175)). You can add items to the List window in several ways.

Adding items with drag and drop

You can drag and drop items into the List window from the Signals, Source, Process, Variables, Wave, or Structure window. Select the items in the first window, then drop them into the List window. Depending on what you select, all items or any portion of the design may be added.

Adding items from the Main window command line

Invoke the **add list** (CR-32) command to add one or more individual items; separate the names with a space:

```
add list <item_name> <item_name>
```

You can add all the items in the current region with this command:

```
add list *
```

Or add all the items in the design with:

```
add list -r /*
```

Adding items with a List window format file

To use a List window format file you must first save a format file for the design you are simulating. The saved format file can then be used as a DO file to recreate the List window formatting. Follow these steps:

- Add HDL items to your List window.
- Edit and format the items to create the view you want (see "Editing and formatting HDL items in the List window" (UM-172)).
- Save the format to a file by selecting **File > Save Format** (List window).

To use the format file, start with a blank List window, and run the DO file in one of two ways:

- Invoke the **do** (CR-68) command from the command line:
  `do <my_list_format>`
- Select **File > Load Format** from the List window menu bar.

Note: List window format files are design-specific; use them only with the design you were simulating when they were created. If you try to use the wrong format file, ModelSim will advise you of the HDL items it expects to find.

The List window menu bar

The following menu commands are available from the List window menu bar.

File menu

Open Dataset	open an existing WLF file
Save Dataset	save data from the current simulation to a WLF file
Write List	save the List window data to a text file in one of three formats; see "Saving List window data to a file" (UM-179) for details
Save Format	save the current List window display and signal preferences to a DO (macro) file; running the DO file will reformat the List window to match the display as it appeared when the DO file was created
Load Format	run a List window format DO file previously saved with Save Format
Close	close this copy of the List window

Edit menu

Cut	cut the selected item field from the listing; see "Editing and formatting HDL items in the List window" (UM-172)
Copy	copy the selected item field
Paste	paste the previously cut or copied item to the left of the currently selected item
Delete	delete the selected item field
Select All	select all signals in the List window
Unselect All	deselect all signals in the List window
Add Marker	add a time marker at the currently selected line
Delete Marker	delete the selected marker from the listing
Find	find the specified item label within the List window

View menu

Signal Properties	set label, radix, trigger on/off, and field width for the selected item
Goto	choose the time marker to go to from a list of current markers

Tools menu

Combine Signals	combine the selected fields into a user-defined bus; keep copies of the original items rather than moving them; see "Combining items in the List window" (UM-174)
Window Preferences	set display properties for all items in the window: delta settings, trigger on selection, strobe period, label size, and dataset prefix

Window menu

The Window menu is identical in all windows. See "Window menu" (UM-144) for a description of the commands.

Editing and formatting HDL items in the List window

Once you have the HDL items you want in the List window, you can edit and format the list to create the view you find most useful. (See also, "Adding HDL items to the List window" (UM-169))

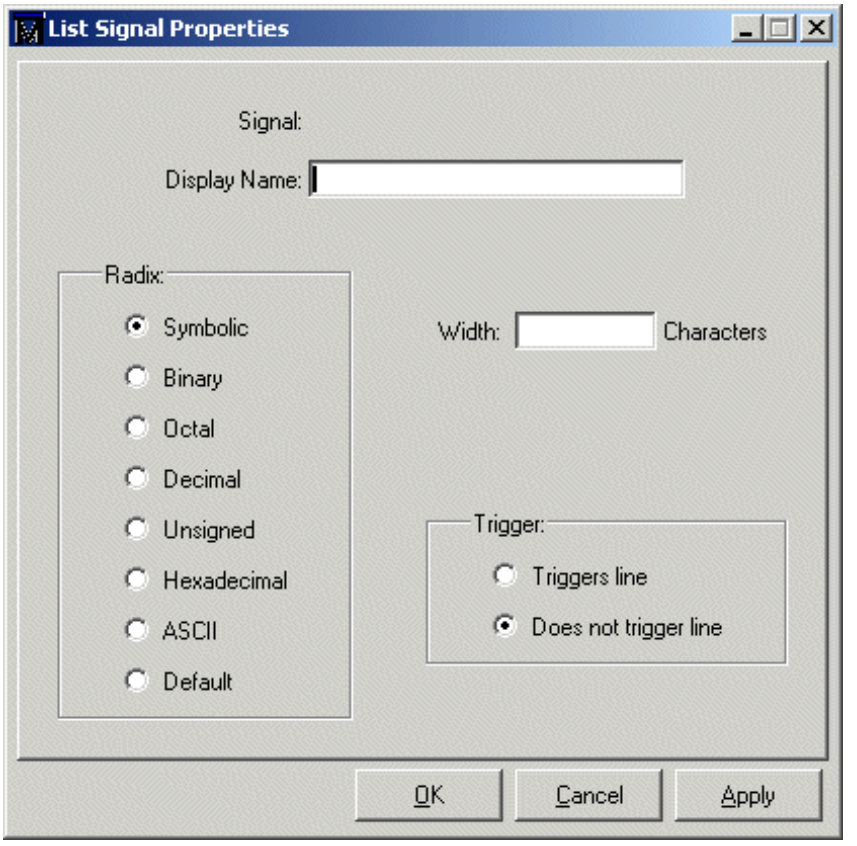
To edit an item:

Select the item's label at the top of the List window or one of its values from the listing. Move, copy or remove the item by selecting commands from the List window Edit menu (UM-170) menu.

You can also click+drag to move items within the window.

To format an item:

Select the item's label at the top of the List window or one of its values from the listing, then select **View > Signal Properties** (List window). The resulting List Signal Properties dialog box allows you to set the item's label, label width, triggering, and radix.



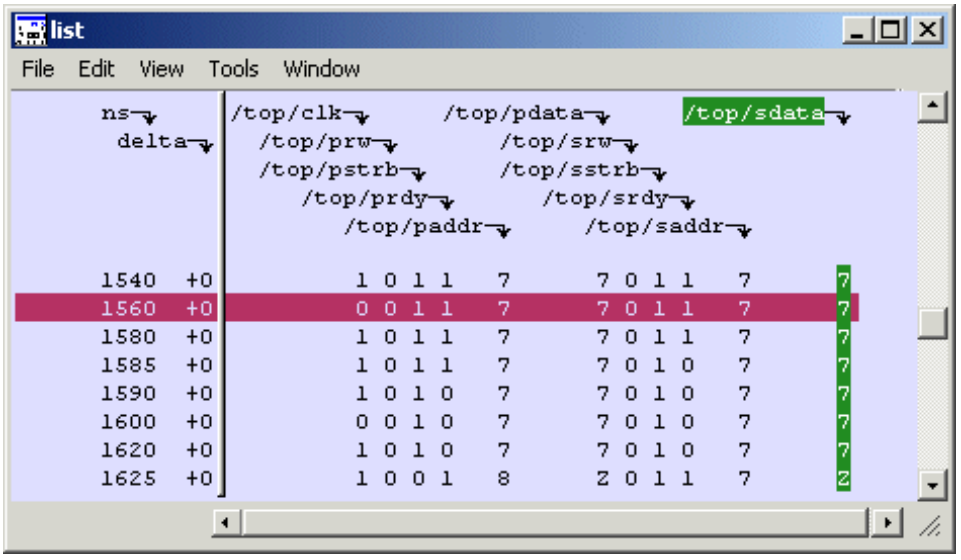
The **List Signal Properties** dialog box includes these options:

- **Signal**
  Shows the full pathname of the selected signal.
- **Display Name**
  Specifies the label that appears at the top of the List window column.

- **Radix**
  Specifies the radix (base) in which the item value is expressed. The default radix is symbolic, which means that for an enumerated type, the List window lists the actual values of the enumerated type of that item. You can change the default radix for the current simulation using either **Simulate > Simulation Options** (Main window) or the **radix** command (CR-108). You can change the default radix permanently by editing the DefaultRadix (UM-345) variable in the *modelsim.ini* file.

  For the other radixes - binary, octal, decimal, unsigned, hexadecimal, or ASCII - the item value is converted to an appropriate representation in that radix. In the system initialization file, *modelsim.tcl*, you can specify the list translation rules for arrays of enumerated types for binary, octal, decimal, unsigned decimal, or hexadecimal item values in the design unit.

  Changing the radix can make it easier to view information in the List window. Compare the image below (with decimal values) with the image on page UM-168 (with symbolic values).

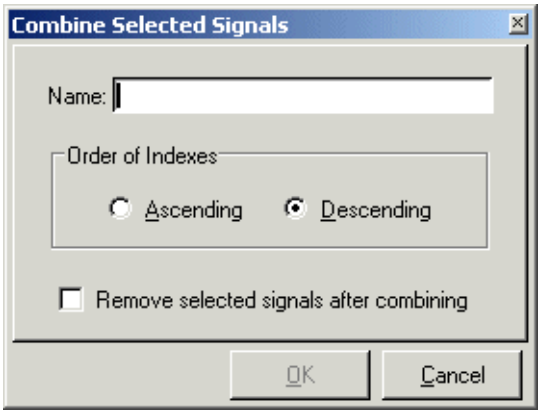


- **Width**
  Allows you to specify the desired width of the column used to list the item value. The default is an approximation of the width of the current value.
- **Trigger: Triggers line**
  Specifies that a change in the value of the selected item causes a new line to be displayed in the List window.
- **Trigger: Does not trigger line**
  Specifies that a change in the value of the selected item does not affect the List window.

  The trigger specification affects the trigger property of the selected item. See also, "Setting List window display properties" (UM-175).

Combining items in the List window

You can combine signals in the List window into busses. A bus is a collection of signals concatenated in a specific order to create a new virtual signal with a specific value. To create a bus, select one or more signals in the List window and then choose **Tools > Combine Signals**.



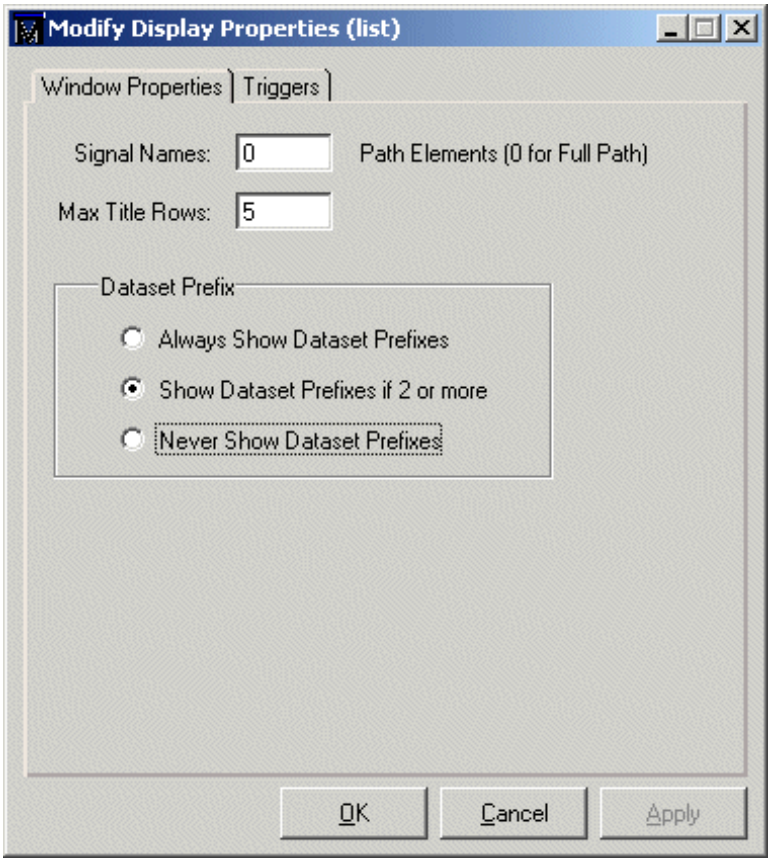
The **Combine Selected Signals** dialog box includes these options:

- **Name**
  Specifies the name of the newly created bus.
- **Order of Indexes**
  Specifies in which order the selected signals are indexed in the bus. If set to **Ascending**, the first signal selected in the List window will be assigned an index of 0. If set to **Descending**, the first signal selected will be assigned the highest index number. Note that the signals are added to the bus in the order that they appear in the window. Ascending and descending affect only the order and direction of the indexes of the bus.
- **Remove selected signals after combining**
  Specifies whether you want to remove the selected signals from the List window once the bus is created.

Setting List window display properties

Before you add items to the List window you can set the window's display properties. To change when and how a signal is displayed in the List window, select **Tools > Window Preferences** (List window). The resulting Modify Display Properties dialog box contains tabs for Window Properties and Triggers.

Window Properties tab



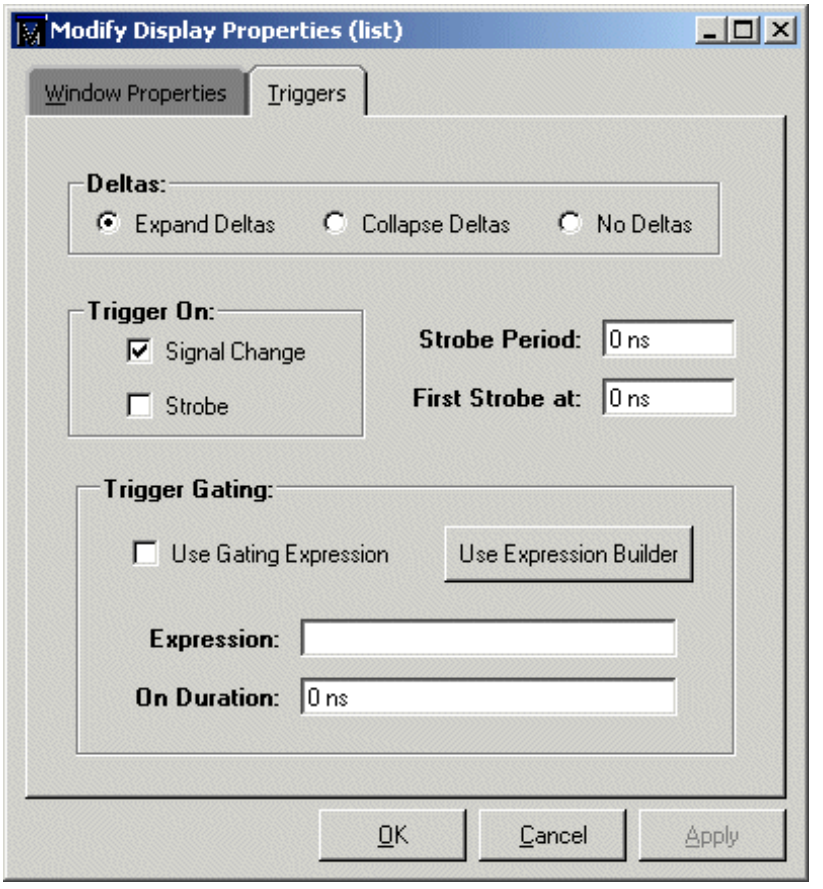
The **Window Properties** tab includes these options:

- **Signal Names**
  Sets the number of path elements to be shown in the List window. For example, "0" shows the full path. "1" shows only the leaf element.
- **Max Title Rows**
  Sets the maximum number of rows in the name pane.
- **Dataset Prefix: Always Show Dataset Prefixes**
  Displays the dataset prefix associated with each signal pathname. Useful for displaying signals from multiple datasets.
- **Dataset Prefix: Show Dataset Prefix if 2 or more**
  Displays dataset prefixes if there are signals in the window from 2 or more datasets.

- **Dataset Prefix: Never Show Dataset Prefixes**
  Turns off display of dataset prefixes.

Trigger settings tab

The **Triggers** tab controls the triggering for the display of new lines in the List window. You can specify whether an HDL item trigger or a strobe trigger is used to determine when the List window displays a new line. If you choose **Trigger on: Signal Change**, then you can choose between collapsed or expanded delta displays. You can also choose a combination of signal and strobe triggers. To use gating, **Signal Change** or **Strobe** or both must be selected. See "Configuring a List trigger with Expression Builder" (UM-382) for an example.



The **Triggers** tab includes the following options:

- **Expand Deltas**
  When selected with the **Trigger on: Signal Change** check box, displays a new line for each time step on which items change, including deltas within a single unit of time resolution.
- **Collapse Deltas**
  Displays only the final value for each time unit.
- **No Deltas**
  Hides simulation cycle (delta) column.

- **Trigger On Signal Change**
  Triggers on signal changes. Defaults to all signals. Individual signals can be excluded from triggering by using the **View > Signal Properties** dialog box or by originally adding them with the **-notrigger** option to the **add list** command (CR-32).
- **Trigger On Strobe**
  Triggers on the **Strobe Period** you specify; specify the first strobe with **First Strobe at:**.
- **Use Gating Expression**
  Enables triggers to be gated on (a value of 1) or off (a value of 0) by the specified
  ExpressionOn Duration
  The duration for gating to remain open after the last list row in which the expression evaluates to true; expressed in x number of default timescale units. Gating is level-sensitive rather than edge-triggered.

Finding items by name in the List window

The Find dialog box allows you to search for text strings in the List window. Select **Edit > Find** (List window) to bring up the Find dialog box.

Enter a text string and **Find** it by searching **Right** or **Left** through the List window display. Specify **Name** to search the real pathnames of the items or **Label** to search their assigned names (see "Setting List window display properties" (UM-175)).

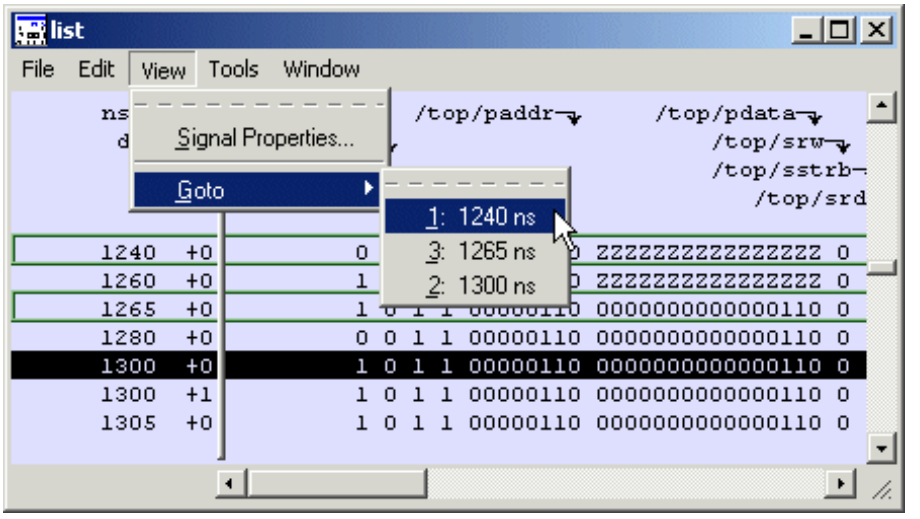
Check **Exact** if you only want to find items that match your search exactly. For example, searching for "clk" without **Exact** will find */top/clk* and *clk1*.

Check **Auto Wrap** to continue the search at the beginning of the window.

Setting time markers in the List window

Select **Edit > Add Marker** (List window) to tag the selected list line with a marker. The marker is indicated by a thin box surrounding the marked line. The selected line uses the same indicator, but its values are highlighted. Delete markers by first selecting the marked line, then selecting **Edit > Delete Marker**.

Finding a marker



Choose a specific marked line to view by selecting **View > Goto**. The marker name (on the **Goto** list) corresponds to the simulation time of the selected line.

Saving List window data to a file

Select **File** > **Write List** (List window) to save the List window data in one of these formats:

- **Tabular**
  writes a text file that looks like the window listing

```
ns   delta      /a       /b      /cin     /sum    /cout
0      +0        X        X        U        X        U
0      +1        0        1        0        X        U
2      +0        0        1        0        X        U
```

- **Events**
  writes a text file containing transitions during simulation

```
@0 +0
/a X
/b X
/cin U
/sum X
/cout U
@0 +1
/a 0
/b 1
/cin 0
```

- **TSSI**
  writes a file in standard TSSI format; see also, the **write tssi** command (CR-222)

```
0 000000000000000010????????
2 000000000000000010???????1?
3 000000000000000010??????010
4 000000000000000010000000010
100 000000010000000010000000010
```

You can also save List window output using the **write list** command (CR-218).

List window keyboard shortcuts

Using the following keys when the mouse cursor is within the List window will cause the indicated actions:

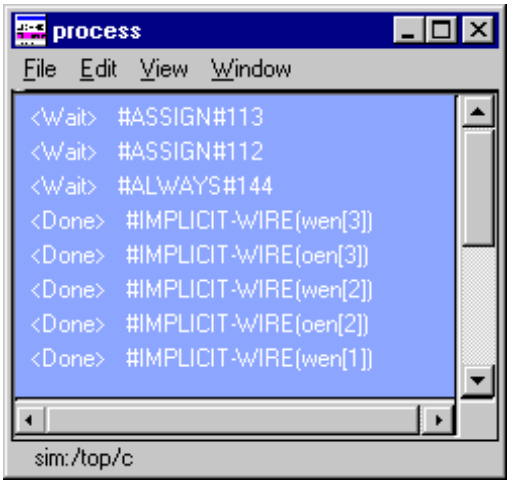
Key	Action
<left arrow>	scroll listing left (selects and highlights the item to the left of the currently selected item)
<right arrow>	scroll listing right (selects and highlights the item to the right of the currently selected item)
<up arrow>	scroll listing up
<down arrow>	scroll listing down
<page up> <control-up arrow>	scroll listing up by page
<page down> <control-down arrow>	scroll listing down by page
<tab>	searches forward (down) to the next transition on the selected signal
<shift-tab>	searches backward (up) to the previous transition on the selected signal (does not function on HP workstations)
<shift-left arrow> <shift-right arrow>	extends selection left/right
<control-f>	opens the Find dialog box to find the specified item label within the list display

Process window

Note: In ModelSim versions 5.7 and later the information contained in the Process window can also be displayed in the Main window Workspace (UM-138). Select **View** > **Active Process** (Main window) when running a simulation.

The Process window displays a list of processes. If **View** > **Active** is selected then all processes scheduled to run during the current simulation cycle are displayed along with the pathname of the instance in which each process is located. If **View** > **In Region** is selected then only the processes in the currently selected region are displayed.

Each HDL item in the scrollbox is preceded by one of the following indicators:



- **<Ready>**
  Indicates that the process is scheduled to be executed within the current delta time.
- **<Wait>**
  Indicates that the process is waiting for a VHDL signal or Verilog net or variable to change or for a specified time-out period.
- **<Done>**
  Indicates that the process has executed a VHDL wait statement without a time-out or a sensitivity list. The process will not restart during the current simulation run.

If you select a "Ready" process, it will be executed next by the simulator.

When you click on a process in the Process window, the following windows are updated:

Window updated	Result
Dataflow window (UM-149)	highlights the selected process
Signals window (UM-183)	shows the signals in the region in which the process is located
Source window (UM-191)	shows the associated source code
Structure window (UM-199)	shows the region in which the process is located
Variables window (UM-203)	shows the VHDL variables and Verilog registers and variables in the process

The Process window menu bar

The following menu commands are available from the Process window menu bar.

File menu

Save List	save the process tree to a text file viewable with the ModelSim **notepad** (CR-95)
Environment	**Follow Context Selection**: update the window based on the selection in the Structure window (UM-199); **Fix to Current Context**: maintain the current view, do not update
Close	close this copy of the Process window

Edit menu

Copy	copy the selected process' full name
Select All	select all processes in the Process window
Unselect All	deselect all processes in the Process window
Find	find the specified text string within the process list; choose the Status (ready, wait or done), the Process label, or the path to search, and the search direction: down or up

View menu

Active	display all the processes that are scheduled to run during the current simulation cycle
In Region	display any processes that exist in the region that is selected in the Structure window
Sort	sort the process list in either ascending, descending, or declaration order

Window menu

The Window menu is identical in all windows. See "Window menu" (UM-144) for a description of the commands.

Signals window

The Signals window is divided into two panes. The left pane shows the names of HDL items in the current region (which is selected in the Structure window). The right pane shows the values of the associated HDL items at the end of the current run. The data in this pane is similar to that shown in the Wave window (UM-206), except that the values do not change dynamically with movement of the selected Wave window cursor.

You can double-click a signal and it will highlight that signal in the Source window (opening a Source window if one is not open already). You can also right click a signal name, and add it to the List, or Wave windows or the current log file.

Horizontal scroll bars for each window pane allow scrolling to the right or left in each pane individually. The vertical scroll bar will scroll both panes together.

The HDL items can be sorted in ascending, descending, or declaration order.

HDL items you can view

One entry is created for each of the following VHDL and Verilog items within the design:

VHDL items

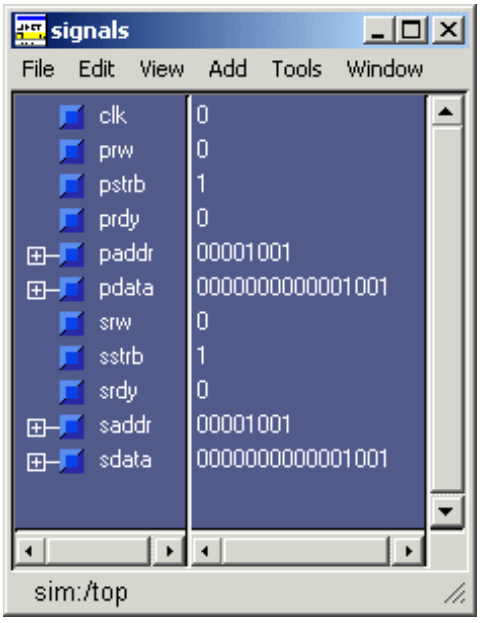
signals, generics, shared variables

Verilog items

nets, registers, variables, named events, and module parameters

Virtual items

(indicated by an orange diamond icon) virtual signals and virtual functions; see "Virtual signals" (UM-125) for more information



VHDL composite types (arrays and record types) and Verilog vector nets, vector registers, and memories are shown in a hierarchical fashion. ModelSim indicates hierarchy with plus (expandable), minus (expanded), and blank (single level) boxes. See "Tree window hierarchical view" (UM-135) for more information.

The Signals window menu bar

The following menu commands are available from the Signals window menu bar.

File menu

Save List	save the signals tree to a text file viewable with the ModelSim **notepad** (CR-95)
Environment	allow the window contents to change based on the current environment; or, fix to a specific context or dataset
Close	close this copy of the Signals window

Edit menu

Copy	copy the current selection in the Signals window
Select All	select all items in the Signals window
Unselect All	unselect all items in the Signals window
Expand Selected	expand the hierarchy of the selected items
Collapse Selected	collapse the hierarchy of the selected items
Expand All	expand the hierarchy of all items that can be expanded
Collapse All	collapse the hierarchy of all expanded items
Force	apply stimulus to the specified Signal Name; specify Value, Kind (Freeze/Drive/Deposit), Delay, and Cancel; see also the **force** command (CR-82)
Noforce	remove the effect of any active **force** command (CR-82) on the selected HDL item; see also the **noforce** command (CR-92)
Clock	define clock signals by Signal Name, Period, Duty Cycle, Offset, and whether the first edge is rising or falling, see"Defining clock signals" (UM-189)
Find	find the specified text string within the Signals window; choose the Name or Value field to search and the search direction: down or up

View menu

Signal Declaration	open the source file in the Source window and highlight the signal declaration
Sort	sort the signals tree in either ascending, descending, or declaration order
Justify Values	justify values to the left or right margins of the window pane
Filter	choose the port and signal types to view (Input Ports, Output Ports, InOut Ports and Internal Signals) in the Signals window

Add menu

Wave	place the Selected Signals, Signals in Region, or Signals in Design in the Wave window (UM-206)
List	place the Selected Signals, Signals in Region, or Signals in Design in the List window (UM-168)
Log	place the Selected Signals, Signals in Region, or Signals in Design in the WLF file

Tools menu

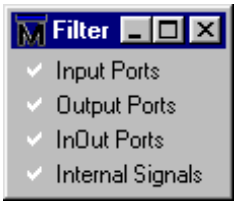
Breakpoints	open the Breakpoints dialog; see "Creating and managing breakpoints" (UM-258)

Window menu

The Window menu is identical in all windows. See "Window menu" (UM-144) for a description of the commands.

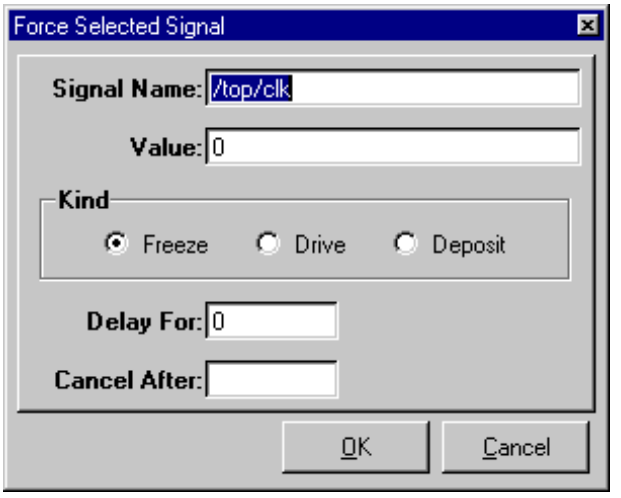
Filtering the signal list

The **View > Filter** menu allows you to specify which HDL items are shown in the Signals window. Multiple options can be selected.



Forcing signal and net values

The **Edit > Force** command displays a dialog box that allows you to apply stimulus to the selected signal or net. Multiple signals can be selected and forced; the force dialog box remains open until all of the signals are either forced, skipped, or you close the dialog box. To cancel a force command, use the **Edit > NoForce** command. See also the **force** command (CR-82).



The **Force** dialog box includes these options:

- **Signal Name**
  Specifies the signal or net for the applied stimulus.
- **Value**
  Initially displays the current value, which can be changed by entering a new value into the field. A value can be specified in radixes other than decimal by using the form (for VHDL and Verilog, respectively):

  ```
  base#value -or- b|o|d|h'value
  ```

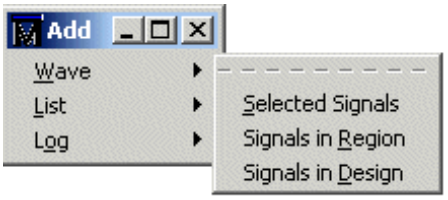
  16#EE or h'EE, for example, specifies the hexadecimal value EE.
- **Kind: Freeze**
  Freezes the signal or net at the specified value until it is forced again or until it is unforced with a **noforce** command (CR-92).

  Freeze is the default for Verilog nets and unresolved VHDL signals and **Drive** is the default for resolved signals.

  If you prefer **Freeze** as the default for resolved and unresolved signals, you can change the default force kind in the *modelsim.ini* file; see *Appendix A - ModelSim variables*.
- **Kind: Drive**
  Attaches a driver to the signal and drives the specified value until the signal or net is forced again or until it is unforced with a **noforce** command (CR-92). This type of force is illegal for unresolved VHDL signals.
- **Kind: Deposit**
  Sets the signal or net to the specified value. The value remains until there is a subsequent driver transaction, or until the signal or net is forced again, or until it is unforced with a **noforce** command (CR-92).

- **Delay For**
  Allows you to specify how many time units from the current time the stimulus is to be applied.
- **Cancel After**
  Cancels the **force** command (CR-82) after the specified period of simulation time.
- **OK**
  When you click the OK button, a **force** command (CR-82) is issued with the parameters you have set, and is echoed in the Main window. If more than one signal is selected to force, the next signal down appears in the dialog box each time the OK button is selected. Unique force parameters can be set for each signal.

Adding HDL items to the Wave and List windows or a WLF file



Use the **Add** menu to add items from the Signals window to the Wave window (UM-206), List window (UM-168), or log file (WLF file). You can also access these same commands by right-clicking a signal in the window.

The WLF file is written as an archive file in binary format and is used to drive the List and Wave windows at a later time. Once signals are added to the WLF file they cannot be removed. If you begin a simulation by invoking **vsim** (CR-189) with the **-view <WLF_fileame>** argument, ModelSim reads the WLF file to drive the Wave and List windows.

Choose one of the following options from the **Add** sub-menus:

- **Selected Signals**
  Adds only the item(s) selected in the Signals window.
- **Signals in Region**
  Adds all items in the region that is selected in the Structure window.
- **Signals in Design**
  Adds all items in the design.

Adding items from the Main window command line

Another way to add items to the Wave or List window or the WLF file is to enter the one of the following commands at the VSIM prompt (choose either the **add list** (CR-32), **add wave** (CR-35), or **log** (CR-87) command):

```
add list | add wave | log <item_name> <item_name>
```

You can add all the items in the current region with this command:

```
add list | add wave | log *
```

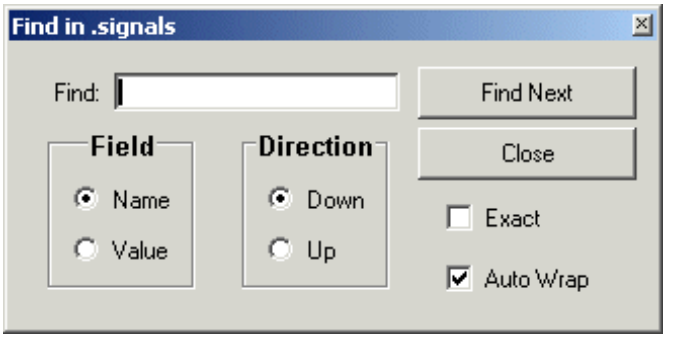
Or add all the items in the design with:

```
add list | add wave | log -r /*
```

If the target window (Wave or List) is closed, ModelSim opens it when you when you invoke the command.

Finding HDL items in the Signals window

To find the specified text string within the Signals window, choose the **Name** or **Value** field to search and the search direction: **Down** or **Up**.



Check **Exact** if you only want to find items that match your search exactly. For example, searching for "clk" without **Exact** will find */top/clk* and *clk1*.

Check **Auto Wrap** to continue the search at the beginning of the window.

You can also do a quick find from the keyboard. When the Signals window is active, each time you type a letter the signal selector (highlight) will move to the next signal whose name begins with that letter.

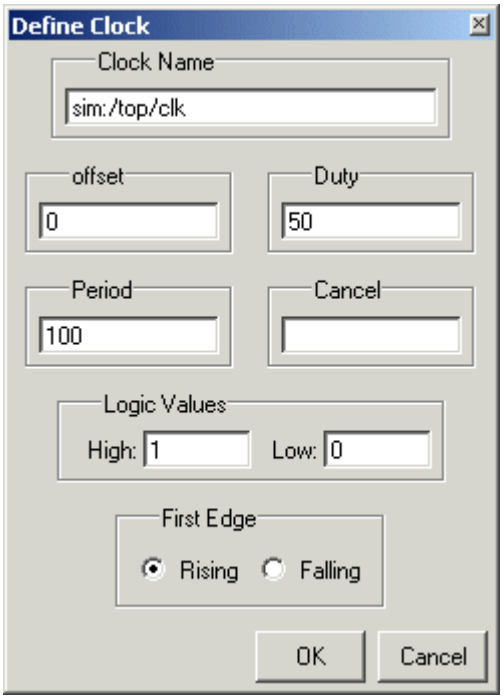
Setting signal breakpoints

You can set "Signal breakpoints" (UM-258) in the Signal window. When a signal breakpoint is hit, a message appears in the Main window Transcript stating which signal caused the breakpoint.

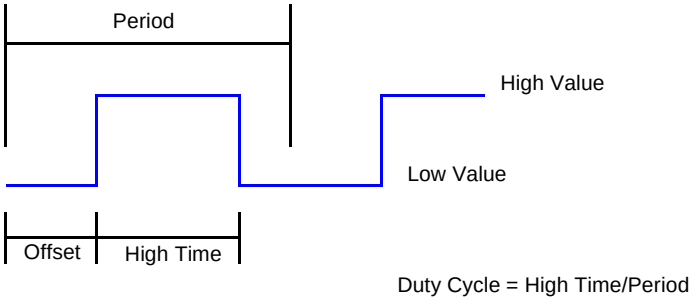
To insert a signal breakpoint, select a signal, click your right mouse button , and select **Insert Breakpoint**. See "Creating and managing breakpoints" (UM-258) for more information.

Defining clock signals

Select **Edit > Clock** to define clock signals by Name, Period, Duty Cycle, Offset, and whether the first edge is rising or falling. You can also specify a simulation period after which the clock definition should be cancelled.



For clock signals starting on the rising edge, the definition for Period, Offset, and Duty Cycle is as follows:



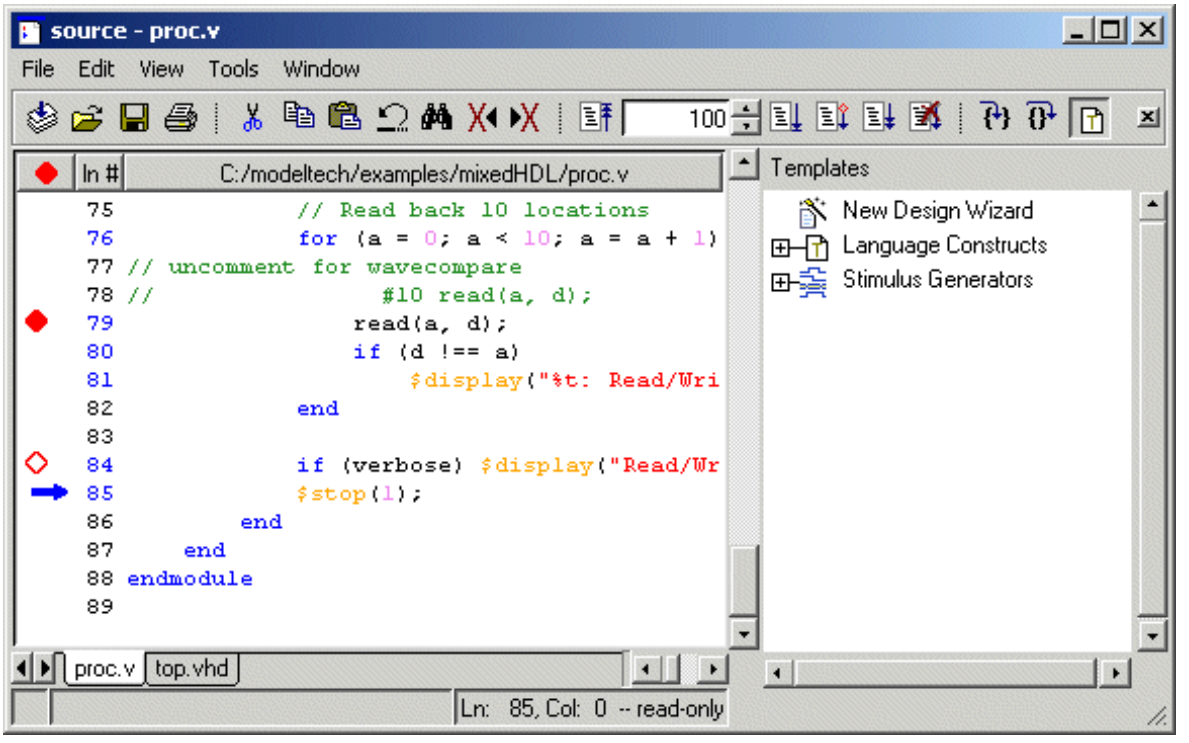
If the signal type is std_logic, std_ulogic, bit, verilog wire, verilog net, or any other logic type where 1 and 0 are valid, then 1 is the default High Value and 0 is the default Low Value. For other signal types, you will need to specify a High Value and a Low Value for the clock.

Source window

The Source window allows you to view and edit your HDL source code. When you first load a design, the source file will display automatically if the Source window is open. Alternatively, you can select an item in a Structure tab of the Main window or use the **File > Open** command (Source window) to add a file to the window..

The window displays your source code with line numbers. As shown in the picture below, you may also see the following:

- Blue line numbers – denote lines on which you can set a breakpoint
- Blue arrow – denotes a process that you have selected in the Process window (UM-181)
- Red diamonds – denote file-line breakpoints; hollow diamonds denote breakpoints that are currently disabled
- File tabs representing each open file
- Templates pane – displays Language templates (UM-264)



Note that files open by default in read-only mode. You can toggle this mode by selecting **Edit > read only**.

The Source window menu bar

The following menu commands are available from the Source window menu bar.

File menu

New	edit a new (VHDL, Verilog or Other) source file
Open	select a source file to open
Open Design Source	open a dialog that lists all source files for the current design
Close File	close the active source file
Use Source	specify an alternative file to use for the current source file; this alternative source mapping exists for the current simulation only
Source Directory	add to a list of directories to search for source files; you can set this permanently using the **SourceDir** variable in the *modelsim.tcl* file
Save	save the current source file
Save As	save the current source file with a different name
Print	print the current source file
Close	close the Source window

Edit menu

To edit a source file, make sure **read only** is *not* selected on the Edit menu.

<editing option>	basic editing options include: Undo, Cut, Copy, Paste, Select All, and Unselect All
Clear highlights	clear highlights that result from double-clicking an error message or a line in a Performance Analyzer report
Comment Selected	turn the selected lines into comments by inserting the correct language comment character at the beginning of each line
Uncomment Selected	removes comment characters from the selected lines
Find	find the specified text string or regular expression within the source file; there is an option to match case or search backwards
Find Next	find the next occurrence of a string specified with the Find command
Replace	find the specified text string or regular expression and replace it with the specified text string or regular expression
read only	toggle the read-only status of the current source file

View menu

Show line numbers	toggle line numbers
Show language templates	toggle display of Language templates (UM-264) pane
Properties	list a variety of information about the source file; for example, file type, file size, file modification date

Tools menu

Examine	display the current value of the selected HDL item; same as the **examine** (CR-75) command; the item name is shown in the title bar
Describe	display information about the selected HDL item; same as the **describe** command (CR-66); the item name is shown in the title bar
Compile	compile the currently active HDL source file
Breakpoints	add, edit, or delete file-line and signal breakpoints; see "Creating and managing breakpoints" (UM-258)
Options	set various Source window options; see Options sub-menu below

Options sub-menu

Colorize Source	colorize key words, variables, and comments
Highlight Executable Lines	highlight the line numbers of executable lines
Middle Mouse Button Paste	enable/disable pasting by pressing the middle-mouse button
Verilog Highlighting	specify Verilog-style colorizing
VHDL Highlighting	specify VHDL-style colorizing
Freeze File	maintain the same source file in the Source window (useful when you have two Source windows open; one can be updated from the Structure window (UM-199), the other frozen)
Freeze View	disable updating the source view from the Process window (UM-181)
Auto-Indent Mode	indent code automatically when editing the file
Tab Stops	set tab stop distance in Source window (see "Setting tab stops in the Source window" (UM-198))

Window menu

The Window menu is identical in all windows. See "Window menu" (UM-144) for a description of the commands.

The Source window toolbar

Buttons on the Source window toolbar give you quick access to these ModelSim commands and functions.

Source window toolbar buttons		
Button	**Menu equivalent**	**Other equivalents**
Compile this file open the Compile HDL Source File dialog	Tools > Compile	use **vcom** or **vlog** command at the VSIM prompt see: **vcom** (CR-145) or **vlog** (CR-181) command
Open Source File open the Open File dialog box (you can open any text file for editing in the Source window)	File > Open	select an HDL item in the Structure window, the associated source file is loaded into the Source window

Source window toolbar buttons			
Button		**Menu equivalent**	**Other equivalents**
	Save Source File save the file in the Source window	File > Save	none
	Print prints the current source file	File > Print	none
	Cut cut the selected text within the Source window	Edit > Cut	see: "Mouse and keyboard shortcuts" (UM-147)
	Copy copy the selected text within the Source window	Edit > Copy	see: "Mouse and keyboard shortcuts" (UM-147)
	Paste paste the copied text to the cursor location	Edit > Paste	see: "Mouse and keyboard shortcuts" (UM-147)
	Undo undo the last action	Edit > Undo	<control - z><control - ->
	Find find the specified text string within the source file; match case option	Edit > Find	<control -f>
	Restart reload the design elements and reset the simulation time to zero, with the option of using current formatting, breakpoints, and WLF file	Main window: Simulate > Run > Restart	**restart** <arguments> see: **restart** (CR-111)
	Run Length specify the run length for the current simulation	Main window: Simulate > Simulation Options	**run** <specific run length> see: **run** (CR-114)
	Run run the current simulation for the specified run length	Main window: Simulate > Run <default_run_length>	**run** (no arguments) see: **run** (CR-114)

Source window toolbar buttons			
Button		**Menu equivalent**	**Other equivalents**
	Continue Run continue the current simulation run until the end of specified run length or until it hits a breakpoint or specified break event	Main window: Simulate > Run > Continue	**run -continue** see: **run** (CR-114)
	Run -All run the current simulation forever, or until it hits a breakpoint or specified break event	Main window: Simulate > Run > Run -All	**run -all** see: **run** (CR-114), see "Assertions tab" (UM-255)
	Break stop the current simulation run	Main window: Simulate > Break	none
	Step steps the current simulation to the next HDL statement	Main window: Simulate > Run > Step	**step** (no arguments) see: **step** (CR-122) command
	Step Over HDL statements are executed but treated as simple statements instead of entered and traced line by line	Main window: Simulate > Run > Step -Over	**step -over** see: **step** (CR-122) command
	Show language templates toggle display of language template pane	View > Show Language Templates	none

Setting file-line breakpoints

You can easily set "File-line breakpoints" (UM-258) in the Source window using your mouse. Click on a blue line number at the left side of the Source window, and a red diamond denoting a breakpoint will appear. The breakpoints are toggles – click once to create the colored diamond; click again to disable or enable the breakpoint.

To delete the breakpoint completely, click the red diamond with your right mouse button, and select **Remove Breakpoint**. Other options on the context menu include:

- **Disable/Enable Breakpoint**
  Deactivate or activate the selected breakpoint.
- **Edit Breakpoint**
  Open the **File Breakpoint** dialog to change breakpoint arguments; see "Adding a breakpoint" (UM-260) for a description of the dialog.
- **Edit All Breakpoints**
  Open the **Modify Breakpoints** dialog; see "Breakpoints dialog" (UM-259).

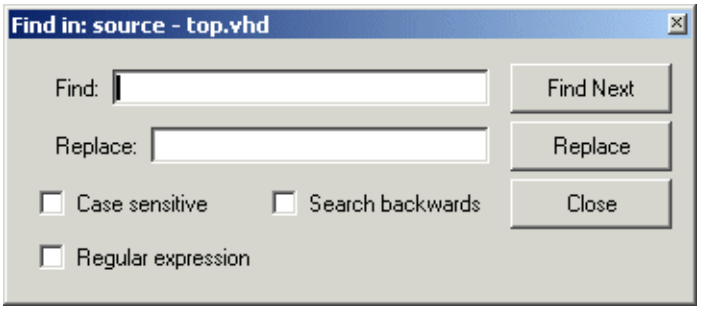
Checking HDL item values and descriptions

There are two quick methods to determine the value and description of an HDL item displayed in the Source window:

- select an item, then choose **Tools > Examine** or **Tools > Describe** from the Source window menu
- pause over an item with your mouse pointer to see an examine pop-up

You can also invoke the **examine** (CR-75) and/or **describe** (CR-66) command on the command line or in a macro.

Finding and replacing in the Source window



The Find dialog box allows you to find and replace text strings or regular expressions in the Source window. Select **Edit > Find** or **Edit > Replace** to bring up the Find dialog box. If you select **Edit > Find**, the **Replace** field is absent from the dialog.

Enter the value to search for in the **Find** field. If you are doing a replace, enter the appropriate value in the **Replace** field. Optionally specify whether the entries are **case sensitive** and whether to **search backwards** from the current cursor location. Check the **Regular expression** checkbox if you are using regular expressions.

Setting tab stops in the Source window

You can set tab stops in the Source window by selecting **Tools** > **Options** > **Tab Stops** or by editing the tabs variable in the **Edit Preferences** dialog.

Follow these steps to set tab stops using the GUI.

1. Select **Tools** > **Options** > **Tab Stops** (Source window).

2. In the dialog that appears, enter either a single number "n" and units, which sets a tab stop every n units, or enter a list of numbers which sets a tab at each location. Available units and their abbreviations are as follows:

Units	Abbreviations
centimeters	c, cm
millimeters	m, mm
inches	i, in
points	p
pixels (screen units)	u
characters	char, chars

If you don’t specify units, they default to characters.

Here are three examples:

- Enter 5 to set a tab stop every 5 characters.
- Enter 10c to set a tab stop every 10 centimeters.
- Enter a list of numbers like the following to set tab stops at specific character locations: 21 49 77 105 133 161 189 217 245 273 301 329 357 385 413 441 469

Important: Do not use quotes or braces in the list (i.e., "21 49" or {21 49}); this will cause the GUI to hang.

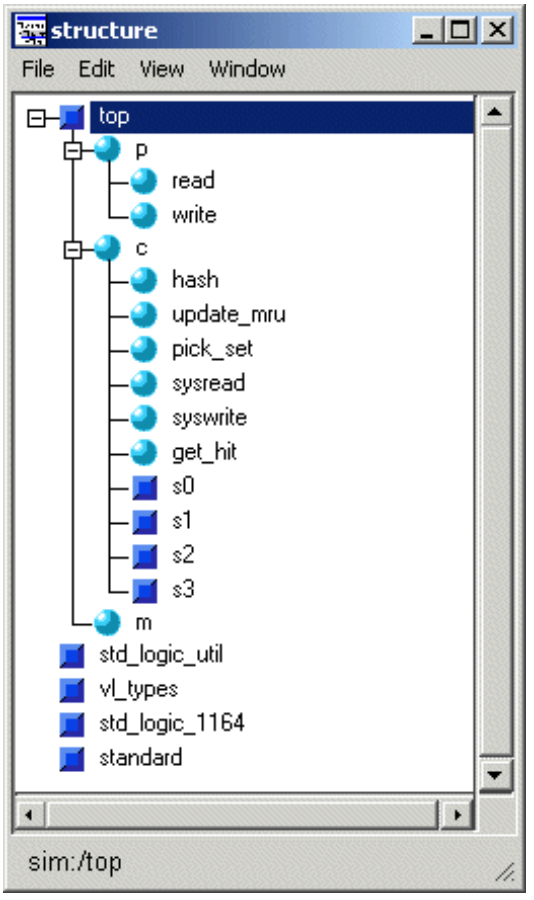
Structure window

Note: In ModelSim versions 5.5 and later, the information contained in the Structure window is shown in the structure tabs of the Main window Workspace (UM-138). The Structure window will not display by default. You can display the Structure window at any time by selecting **View > Structure** (Main window). The discussion below applies to both the Structure window and the structure tabs in the workspace.

The Structure window provides a hierarchical view of the structure of your design. An entry is created by each HDL item within the design.

HDL items you can view

The following HDL items for VHDL and Verilog are represented by hierarchy within the Structure window.



VHDL items

(indicated by a dark blue square icon)
component instantiations, generate statements, block statements, and packages

Verilog items

(indicated by a lighter blue circle icon)
module instantiations, named forks, named begins, tasks, and functions

Virtual items

(indicated by an orange diamond icon)
virtual regions; see "Virtual Objects (User-defined buses, and more)" (UM-125) for more information.

You can expand and contract the display to view the hierarchical structure by clicking on the boxes that contain "+" or "-". Clicking "+" expands the hierarchy so the sub-elements of that item can be seen. Clicking "-" contracts the hierarchy.

The first line of the Structure window indicates the top-level design unit being simulated. By default, this is the only level of the hierarchy that is expanded upon opening the Structure window.

When you select a region in the Structure window, it becomes the *current region* and is highlighted; the Source window (UM-191) and Signals window (UM-183) change dynamically to reflect the information for that region. This feature provides a useful method for finding the source code for a selected region because the system keeps track of the pathname where the source is located and displays it automatically, without the need for you to provide the pathname.

Also, when you select a region in the Structure window, the Process window (UM-181) is updated if **In Region** is selected in that window. The Process window will in turn update the Variables window (UM-203).

Structure window menu bar

The following menu commands are available from the Structure window menu bar. Some of the commands are also available from a context menu in a Structure tab of the Main window workspace.

File menu

Save List	save the structure tree to a text file viewable with the ModelSim **notepad** (CR-95)
Environment	1) specify that the window contents change when the active dataset is changed; 2) fix the window contents to a specific dataset; or 3) change to a new root context
Close	close this copy of the Structure window

Edit menu

Copy	copy the current selection in the Structure window
Expand Selected	expand the hierarchy of the selected item
Collapse Selected	collapse the hierarchy of the selected item
Expand All	expand the hierarchy of all items that can be expanded
Collapse All	collapse the hierarchy of all expanded items
Find	find the specified text string within the structure tree; see "Finding items in the Structure window" (UM-202)

View menu

Sort	sort the structure tree in either ascending, descending, or declaration order

Window menu

The Window menu is identical in all windows. See "Window menu" (UM-144) for a description of the commands.

Structure window context menu

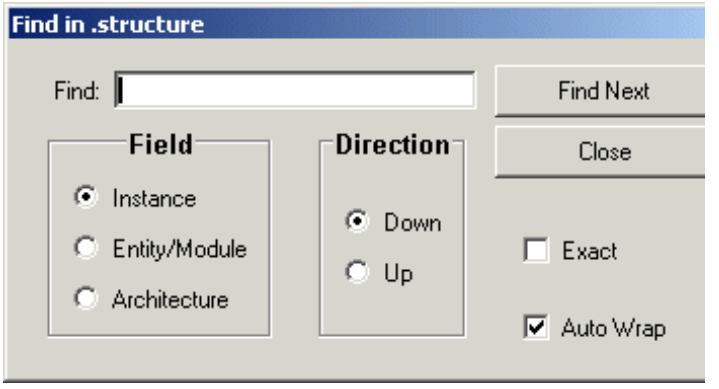
The Structure window has a context menu that you access by clicking the right-mouse button.

The Structure tab context menu includes the following options.

- **View Source**
  Opens the source file in the Source window (UM-191). Double-clicking will also open the source file.
- **Add**
  Add the selected item to the Dataflow, List, or Wave window or to the current Log file.
- **Sort**
  Sorts the HDL items in the Structure tab by alphabetic (ascending or descending) or declaration order.
- **Find**
  Opens the Find dialog. See "Finding items in the Structure window" (UM-202) for details.
- **Expand Selected**
  Shows the hierarchy of the selected HDL item.
- **Collapse Selected**
  Hides the hierarchy of the selected HDL item.
- **Expand All**
  Shows the hierarchy of all HDL items in the list.
- **Collapse All**
  Hides the hierarchy of all HDL items in the list.
- **Save List**
  Writes the HDL item names in the Structure tab to a text file.
- **Save Dataset**
  Saves the current simulation to a WLF file.

- **End Simulation**
  Terminates the active simulation. This command will be Close <dataset name> on a dataset Structure tab.
- **Close <dataset name>**
  Closes the specified dataset.

Finding items in the Structure window



The Find dialog box allows you to search for text strings in the Structure window. Select **Edit > Find** (Structure window) to bring up the Find dialog box.

Enter the value to search for in the **Find** field. Specify whether you are looking for an **Instance**, **Entity/Module**, or **Architecture**. Also specify which direction to search.

Check **Exact** if you only want to find items that match your search exactly. For example, searching for "clk" without **Exact** will find */top/clk* and *clk1*.

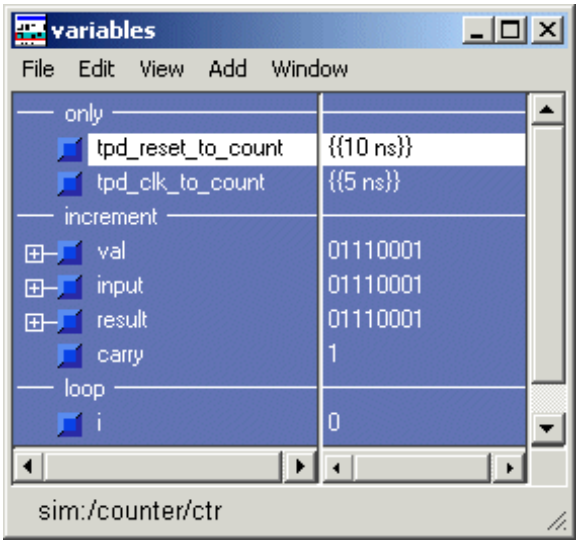
Check **Auto Wrap** to continue the search at the beginning of the window.

Variables window

The Variables window is divided into two window panes. The left pane lists the names of HDL items within the current process. The right pane lists the current value(s) associated with each name. The pathname of the current process is displayed at the bottom of the window.

HDL items you can view

The following HDL items for VHDL and Verilog are viewable within the Variables window.



VHDL items

constants, generics, and variables

Verilog items

registers and variables

VHDL composite types (arrays and record types) and Verilog vector registers and memories are shown in a hierarchical fashion. ModelSim indicates hierarchy with plus (expandable), minus (expanded), and blank (single level) boxes. See "Tree window hierarchical view" (UM-135) for more information.

To change the value of a VHDL variable, constant, or generic or a Verilog register or variable, move the pointer to the desired name and click to highlight the selection. Select **Edit > Change** (Variables window) to bring up a dialog box that lets you specify a new value. You can enter any value that is valid for the variable. An array value must be specified as a string (without surrounding quotation marks). To modify the values in a record, you need to change each field separately.

Click on a process in the Process window to change the Variables window.

The Variables window menu bar

The following menu commands are available from the Variables window menu bar.

File menu

Save List	save the variable tree to a text file viewable with the ModelSim **notepad** (CR-95)
Environment	**Follow Process Selection**: update the window based on the selection in the Process window (UM-181) **Fix to Current Process**: maintain the current view, do not update
Close	close this copy of the Variables window

Edit menu

Copy	copy the selected items in the Variables window
Select All	select all items in the Variables window
Unselect All	deselect all items in the Variables window
Expand Selected	expand the hierarchy of the selected item
Collapse Selected	collapse the hierarchy of the selected item
Expand All	expand the hierarchy of all items that can be expanded
Collapse All	collapse the hierarchy of all expanded items
Change	change the value of the selected HDL item
Find	find the specified text string within the variables tree; choose the Name or Value field to search and the search direction: Down or Up

View menu

Sort	sort the variables tree in either ascending, descending, or declaration order
Justify Values	justify values to the left or right margins of the window pane

Add menu

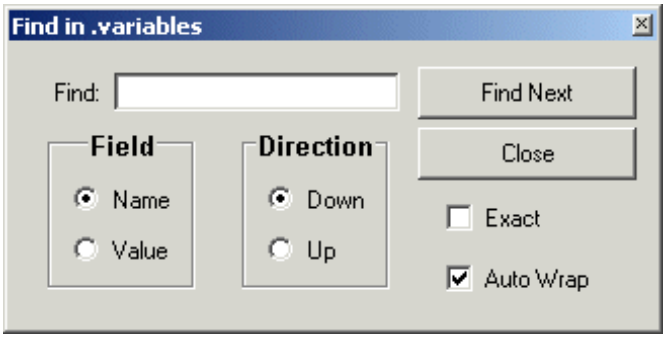
Wave/List/Log	place the Selected Variables or Variables in Region in the Wave window (UM-206), List window (UM-168), or WLF file

Window menu

The Window menu is identical in all windows. See "Window menu" (UM-144) for a description of the commands.

Finding HDL items in the Variables window

To find the specified text string within the Variables window, choose the **Name** or **Value** field to search and the search direction: **Down** or **Up**.



Check **Exact** if you only want to find items that match your search exactly. For example, searching for "clk" without **Exact** will find */top/clk* and *clk1*.

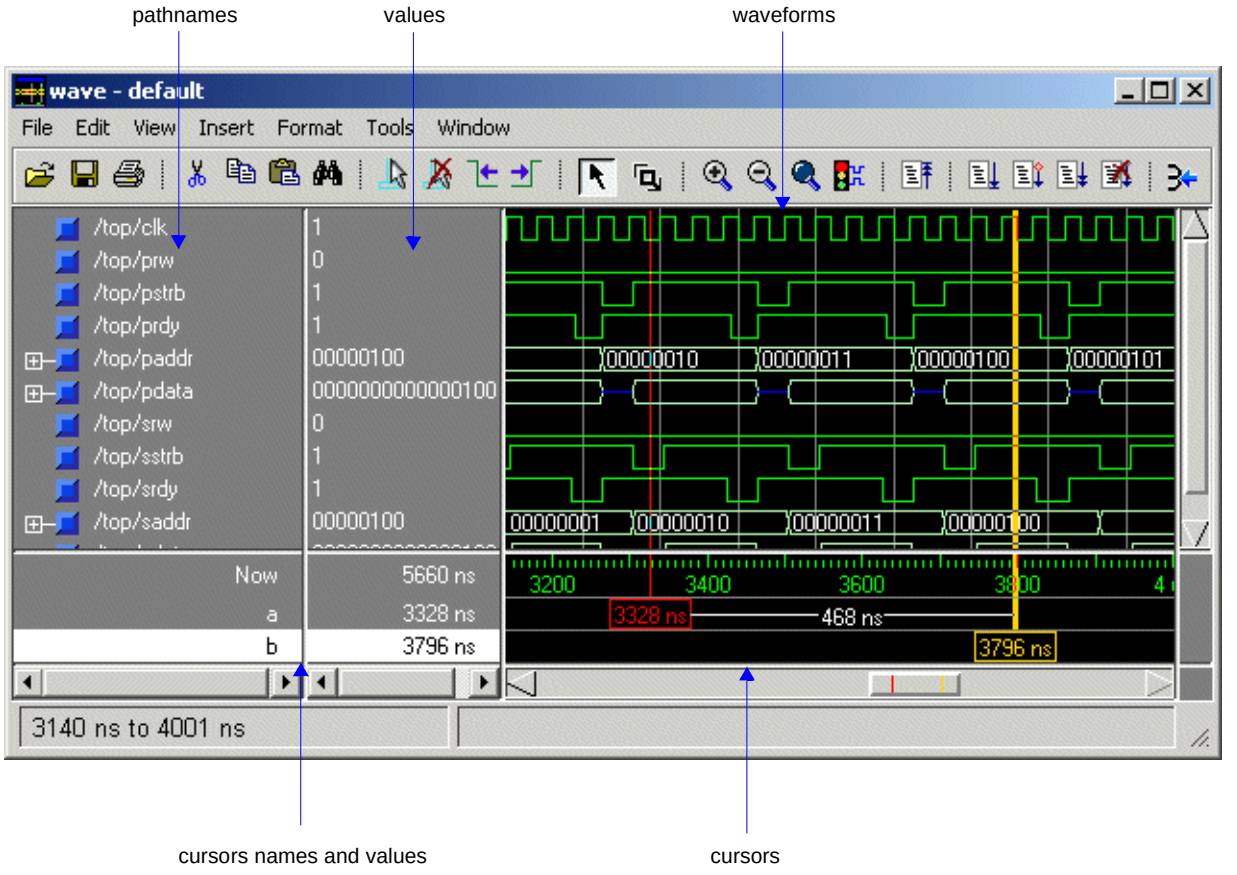
Check **Auto Wrap** to continue the search at the beginning of the window.

You can also do a quick find from the keyboard. When the Variables window is active, each time you type a letter the highlight will move to the next item whose name begins with that letter.

Wave window

The Wave window, like the List window, allows you to view the results of your simulation. In the Wave window, however, you can see the results as HDL waveforms and their values.

The Wave window is divided into a number of window panes. All window panes in the Wave window can be resized by clicking and dragging the bar between any two panes.



Pathname pane

The pathname pane displays signal pathnames. Signals can be displayed with full pathnames, as shown here, or with only the leaf element displayed. You can increase the size of the pane by clicking and dragging on the right border. The selected signal is highlighted.

The white bar along the left margin indicates the selected dataset (see "Splitting Wave window panes" (UM-216)).

Values pane

The values pane displays the values of the displayed signals.

The radix for each signal can be symbolic, binary, octal, decimal, unsigned, hexadecimal, ASCII, or default. The default radix can be set by selecting **Simulate > Simulation Options** (Main window) (see "Setting default simulation options" (UM-254)).

The data in this pane is similar to that shown in the Signals window (UM-183), except that the values change dynamically whenever a cursor in the waveform pane is moved.

Waveform pane

The waveform pane displays the waveforms that correspond to the displayed signal pathnames. It also displays up to 20 cursors. Signal values can be displayed in analog step, analog interpolated, analog backstep, literal, logic, and event formats. Each signal can be formatted individually. The default format is logic.

If you rest your mouse pointer on a signal in the waveform pane, a popup displays with information about the signal. You can toggle this popup on and off in the **Wave Window Properties** dialog (see "Setting Wave window display properties" (UM-222)).

Cursor panes

There are three cursor panes–the left pane shows the cursor names; the middle pane shows the current simulation time and the value for each cursor; and the right pane shows the absolute time value for each cursor and relative time between cursors. Up to 20 cursors can be displayed. See "Using time cursors in the Wave window" (UM-226) for more information.

HDL items you can view

VHDL items

(indicated by a dark blue square)
signals and process and shared variables

Verilog items

(indicated by a light blue circle)
nets, registers, variables, and named events

Virtual items

(indicated by an orange diamond)
virtual signals, buses, and functions, see; "Virtual Objects (User-defined buses, and more)" (UM-125) for more information

Comparison items

(indicated by a yellow triangle)
comparison region and comparison signals; see *Chapter 10 - Waveform Comparison* for more information

Note: Constants, generics, and parameters are not viewable in the List or Wave windows.

The data in the item values pane is very similar to the Signals window, except that the values change dynamically whenever a cursor in the waveform pane is moved.

At the bottom of the waveform pane you can see a time line, tick marks, and a readout of each cursor's position. As you click and drag to move a cursor, the time value at the cursor location is updated at the bottom of the cursor.

You can resize the window panes by clicking on the bar between them and dragging the bar to a new location.

Waveform and signal-name formatting are easily changed via the Format menu (UM-211). You can reuse any formatting changes you make by saving a Wave window format file, see "Adding items with a Wave window format file" (UM-208).

Adding HDL items in the Wave window

Before adding items to the Wave window you may want to set the window display properties (see "Setting Wave window display properties" (UM-222)). You can add items to the Wave window in several ways.

Adding items from the Signals window with drag and drop

You can drag and drop items into the Wave window from the List, Process, Signals, Source, Structure, or Variables window. Select the items in the first window, then drop them into the Wave window. Depending on what you select, all items or any portion of the design can be added.

Adding items from the command line

To add specific HDL items to the window, enter (separate the item names with a space):

```
VSIM> add wave <item_name> <item_name>
```

You can add all the items in the current region with this command:

```
VSIM> add wave *
```

Or add all the items in the design with:

```
VSIM> add wave -r /*
```

Adding items with a Wave window format file

To use a Wave window format file you must first save a format file for the design you are simulating. Follow these steps:

1. Add the items you want in the Wave window with any method shown above.
2. Edit and format the items, see "Editing and formatting HDL items in the Wave window" (UM-219) to create the view you want .
3. Save the format to a file by selecting **File > Save Format** (Wave window).

To use the format file, start with a blank Wave window and run the DO file in one of two ways:

- Invoke the **do** command (CR-68) from the command line:

```
VSIM> do <my_wave_format>
```

- Select **File > Load Format** (Wave window).

Note: Wave window format files are design-specific; use them only with the design you were simulating when they were created.

The Wave window menu bar

The following menu commands and button options are available from the Wave window menu bar. Many of these commands are also available via a context menu by clicking your right mouse button within the Wave window itself.

File menu

Open Dataset	open a dataset
Save Dataset	save the current simulation to a WLF file
Save Format	save the current Wave window display and signal preferences to a DO (macro) file; running the DO file will reformat the Wave window to match the display as it appeared when the DO file was created
Load Format	run a Wave window format (DO) file previously saved with Save Format
Save Image	saves bitmap file of Wave window
Page Setup	configure page setup including paper size, margins, label width, cursors, grid, color, scaling and orientation
Print	send the contents of the Wave window to a selected printer; see "Saving waveforms" (UM-233) for details
Print Postscript	save or print the waveform display as a Postscript file; see "Saving waveforms" (UM-233) for details
Close	close this copy of the Wave window

Edit menu

Cut	cut the selected item and waveform from the Wave window; see "Editing and formatting HDL items in the Wave window" (UM-219)
Copy	copy the selected item and waveform
Paste	paste the previously cut or copied item above the currently selected item
Delete	delete the selected item and its waveform
Edit Cursor	open a dialog to specify the location of the selected cursor
Delete Cursor	delete the selected cursor from the window

Delete Window Pane	delete the selected window pane
Select All Unselect All	select, or unselect, all item names in the pathname pane
Find	find the specified item label within the pathname pane or the specified value within the value pane

View menu

Zoom <selection>	selection: Full, In, Out, Last, or Range to change the waveform display range
Mouse Mode	toggle mouse pointer between Select Mode (click left mouse button to select, drag with middle mouse button to zoom) and Zoom Mode (drag with left mouse button to zoom, click middle mouse button to select)
Signal Declaration	open the source file in the Source window and highlight the signal declaration for the currently selected signal
Cursors	choose a cursor to go to from a list of available cursors
Bookmarks	choose a bookmark to go to from a list of available bookmarks
Goto Time	scroll the Wave window so the specified time is in view; "g" hotkey produces the same result
Sort	sort the top-level items in the pathname pane; sort with full path name or viewed name; use ascending or descending order
Justify Values	justify values to the left or right margins of the window pane
Refresh Display	clear the Wave window, empty the file cache, and rebuild the window from scratch
Properties	set properties for the selected item (use the Format menu to change individual properties)

Insert menu

Divider	insert a divider at the current location
Breakpoint	add a breakpoint on the selected signal; see "Signal breakpoints" (UM-258)
Bookmark	add a bookmark with the current zoom range and scroll location; see "Saving zoom range and scroll position with bookmarks" (UM-229)
Cursor	add a cursor to the waveform pane
Window Pane	split the pathname, values and waveform window panes to provide room for a new waveset

Format menu

Radix	set the selected items' radix
Format	set the waveform format for the selected item – Literal, Logic, Event, Analog
Color	set the color for the selected item from a color palette
Height	set the waveform height in pixels for the selected item

Tools menu

Breakpoints	add, edit, and delete signal breakpoints; see "Creating and managing breakpoints" (UM-258)
Bookmarks	add, edit, delete, and goto bookmarks; see "Saving zoom range and scroll position with bookmarks" (UM-229)
Dataset Snapshot	enable periodic saving of simulation data to a WLF file
Combine Signals	combine the selected items into a user-defined bus
Window Preferences	set various display properties such as signal path length, cursor snap distance, row margin, dataset prefixes, waveform popup, etc.

Window menu

The Window menu is identical in all windows. See "Window menu" (UM-144) for a description of the commands.

The Wave window toolbar

The Wave window toolbar gives you quick access to these ModelSim commands and functions.

Wave window toolbar buttons			
Button		**Menu equivalent**	**Other options**
	Load Wave Format run a Wave window format (DO) file previously saved with Save Format	File > Load Format	**do** wave.do see **do** command (CR-68)
	Save Wave Format save the current Wave window display and signal preferences to a do (macro) file	File > Save Format	none
	Print print a user-selected range of the current Wave window display to a printer or a file	File > Print File > Print Postscript	none
	Cut cut the selected signal from the Wave window	Edit > Cut	right mouse in pathname pane > Cut
	Copy copy the selected signal in the signal-name pane	Edit > Copy	right mouse in pathname pane > Copy
	Paste paste the copied signal above another selected signal	Edit > Paste	right mouse in pathname pane > Paste
	Find find a name or value in the Wave window	Edit > Find	<control-f>
	Add Cursor add a cursor to the center of the waveform pane	Insert > Cursor	right mouse in cursor pane

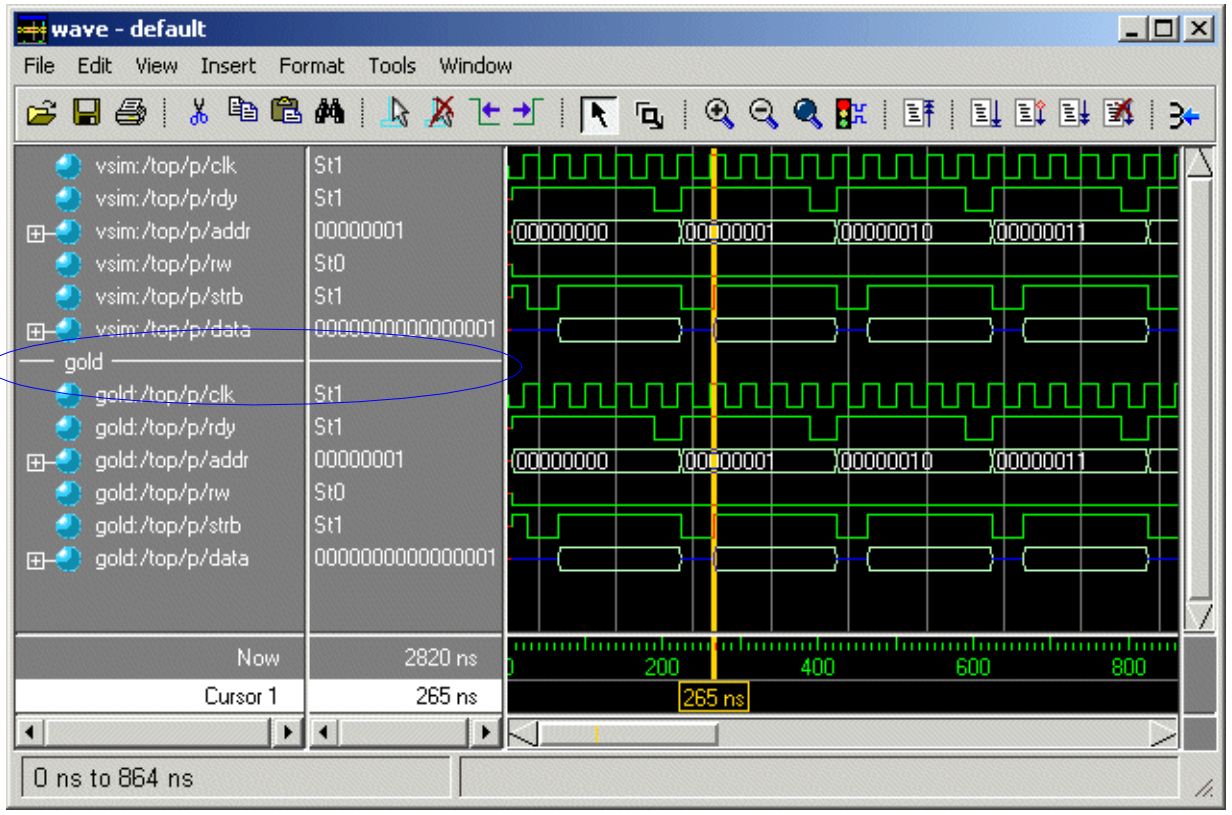
Wave window toolbar buttons			
Button		**Menu equivalent**	**Other options**
	Delete Cursor delete the selected cursor from the window	Edit > Delete Cursor	right mouse in cursor pane > Delete Cursor n
	Find Previous Transition locate the previous signal value change for the selected signal	Edit > Search (Search Reverse)	keyboard: Shift + Tab
	Find Next Transition locate the next signal value change for the selected signal	Edit > Search (Search Forward)	keyboard: Tab
	Select Mode set mouse to Select Mode – click left mouse button to select, drag middle mouse button to zoom	View > Mouse Mode > Select Mode	none
	Zoom Mode set mouse to Zoom Mode – drag left mouse button to zoom, click middle mouse button to select	View > Mouse Mode > Zoom Mode	none
	Zoom in 2x zoom in by a factor of two from the current view	View > Zoom > Zoom In	keyboard: i I or + right mouse in wave pane > Zoom In
	Zoom out 2x zoom out by a factor of two from current view	View > Zoom > Zoom Out	keyboard: o O or - right mouse in wave pane > Zoom Out
	Zoom Full zoom out to view the full range of the simulation from time 0 to the current time	View > Zoom > Zoom Full	keyboard: f or F right mouse in wave pane > Zoom Full
	Stop Wave Drawing halts any waves currently being drawn in the Wave window	none	none

Wave window toolbar buttons			
Button		**Menu equivalent**	**Other options**
	Restart reloads the design elements and resets the simulation time to zero, with the option of keeping the current formatting, breakpoints, and WLF file	Main menu: Simulate > Run > Restart	**restart** <arguments> see: **restart** (CR-111)
	Run run the current simulation for the default time length	Main menu: Simulate > Run > Run <default_length>	use the **run** command at the VSIM prompt see: **run** (CR-114)
	Continue Run continue the current simulation run	Main menu: Simulate > Run > Continue	use the **run -continue** command at the VSIM prompt see: **run** (CR-114)
	Run -All run the current simulation forever, or until it hits a breakpoint or specified break event	Main menu: Simulate > Run > Run -All	use the **run -all** command at the VSIM prompt see: **run** (CR-114), also see "Assertions tab" (UM-255)
	Break stop the current simulation run	none	none
	Show Drivers display driver(s) of the selected signal, net, or register in the Dataflow window	[Dataflow window] Navigate > Expand net to drivers	[Dataflow window] Expand net to all drivers right mouse in wave pane > Show Drivers

Using dividers

Dividers serve as a visual aid to signal debugging, allowing you to separate signals and waveforms for easier viewing. Dividing lines can be placed in the pathname and values window panes by selecting **Insert > Divider** (Wave window). Or, you can add a divider using the **-divider** argument to the **add wave** command (CR-35).

Dividing lines can be assigned any name or no name at all. The default name is "New Divider." In the illustration below, two datasets have been separated with a Divider called "gold." Notice that the waveforms in the waveform window pane have been separated by the divider as well.



After you have added a divider, you can move it, change its properties (name and size), or delete it.

To move a divider — Click and drag the divider to the location you want

To change a divider's name and size — Click the divider with the right mouse button and select Divider Properties from the pop-up menu

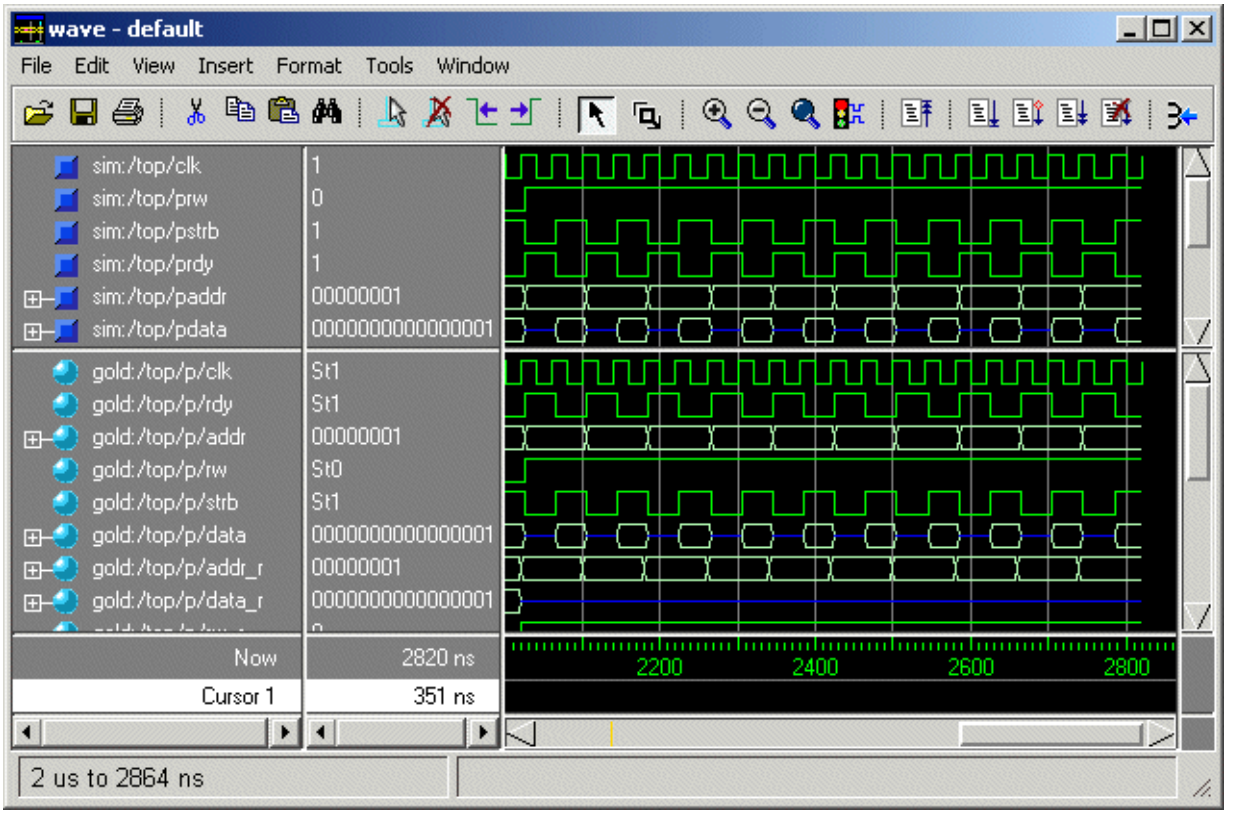
To delete a divider — Select the divider and either press the <Delete> key on your keyboard or select Delete from the pop-up menu

Splitting Wave window panes

The pathnames, values and waveforms window panes of the Wave window display can be split to accommodate signals from one or more datasets. Selecting **Insert > Window Pane** (Wave window) creates a space below the selected dataset and makes the new window pane the selected pane. (The selected wave window pane is indicated by a white bar along the left margin of the pane.)

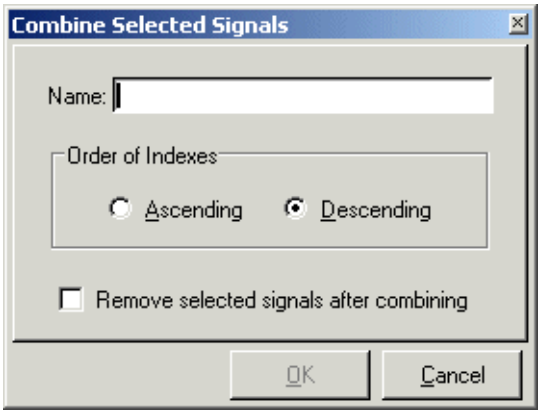
In the illustration below, the Wave window is split, showing the current active simulation with the prefix "sim," and a second view-mode dataset, with the prefix "gold."

For more information on viewing multiple simulations, see *Chapter 6 - WLF files (datasets) and virtuals*.



Combining items in the Wave window

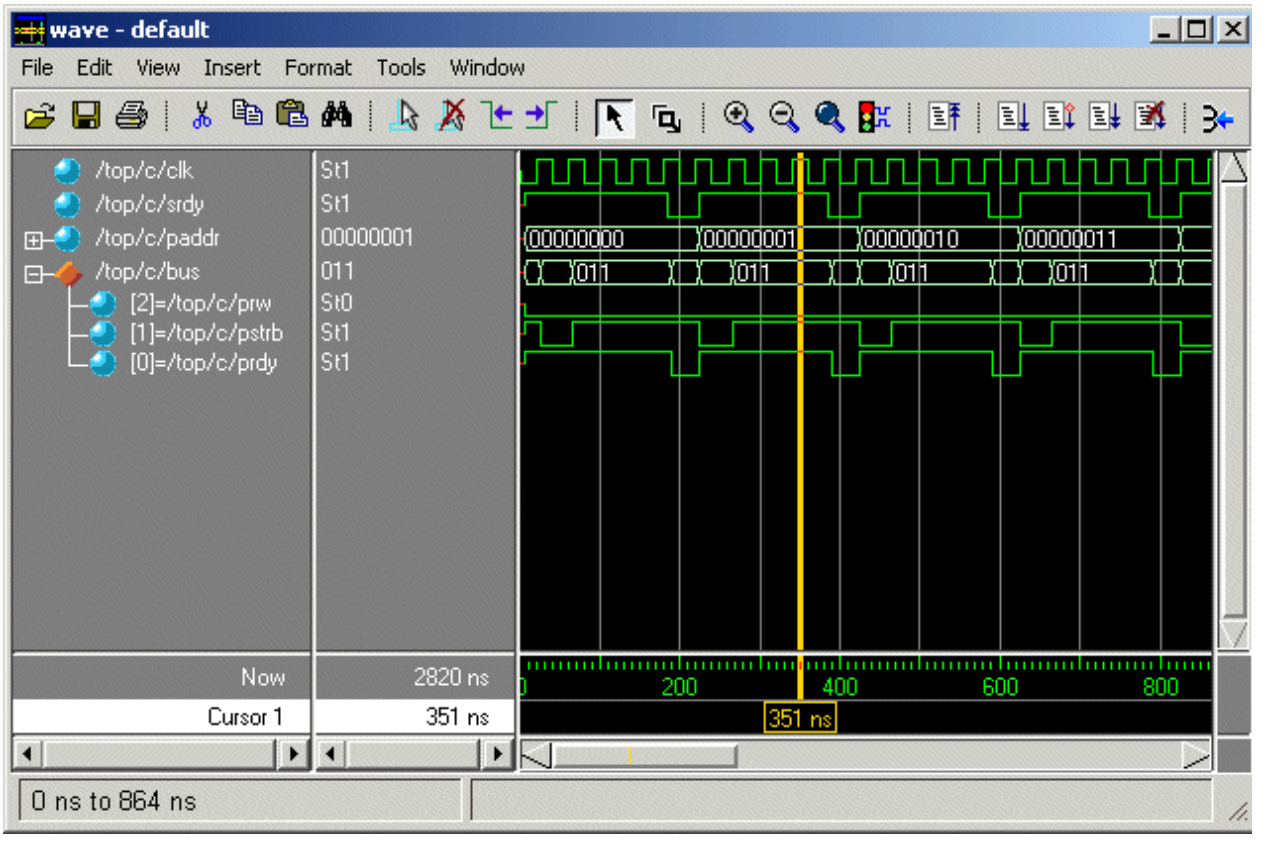
You can combine signals in the Wave window into busses. A bus is a collection of signals concatenated in a specific order to create a new virtual signal with a specific value. To create a bus, select one or more signals in the Wave window and then choose **Tools > Combine Signals**.



The **Combine Selected Signals** dialog box includes these options:

- **Name**
  Specifies the name of the newly created bus.
- **Order of Indexes**
  Specifies in which order the selected signals are indexed in the bus. If set to **Ascending**, the first signal selected in the Wave window will be assigned an index of 0. If set to **Descending**, the first signal selected will be assigned the highest index number. Note that the signals are added to the bus in the order that they appear in the window. Ascending and descending affect only the order and direction of the indexes of the bus.
- **Remove selected signals after combining**
  Specifies whether you want to remove the selected signals from the Wave window once the bus is created.

In the illustration below, three signals have been combined to form a new bus called "bus". Note that the component signals are listed in the order in which they were selected in the Wave window. Also note that the value of the bus is made up of the values of its component signals, arranged in a specific order. Virtual objects are indicated by an orange diamond.



Other virtual items in the Wave window

See "Virtual Objects (User-defined buses, and more)" (UM-125) for information about other virtual items viewable in the Wave window.

Displaying drivers of the selected waveform

You can automatically display in the Dataflow window the drivers of a signal selected in the Wave window. You can do this three ways:

- Select a waveform and click the Show Drivers button on the toolbar.
- Select a waveform and select Show Drivers from the shortcut menu
- Double-click a waveform edge (you can enable/disable this option in the display properties dialog; see "Setting Wave window display properties" (UM-222))

This operation will open the Dataflow window and display the drivers of the signal selected in the Wave window. The Wave pane in the Dataflow window will also open showing the

selected signal with a cursor at the selected time. The Dataflow window will show the signal(s) values at the current time cursor position.

Editing and formatting HDL items in the Wave window

Once you have the HDL items you want in the Wave window, you can edit and format the list in the pathname and values panes to create the view you find most useful. (See also, "Setting Wave window display properties" (UM-222).)

To edit an item:

Select the item's label in the pathname pane or its waveform in the waveform pane. Move, copy, or remove the item by selecting commands from the Wave window **Edit menu** (UM-209).

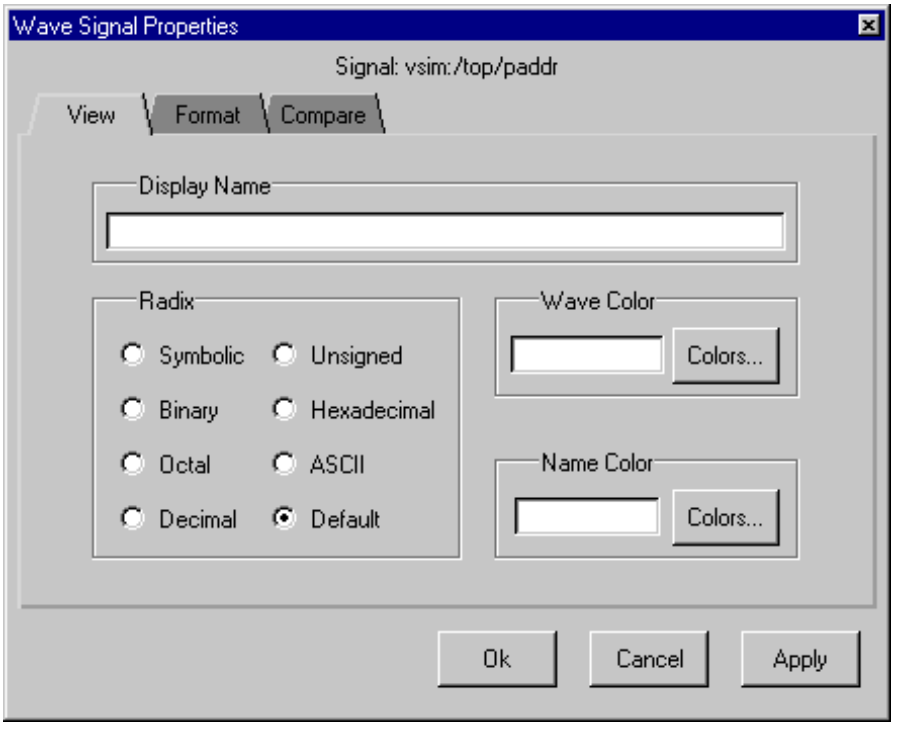
You can also **click+drag** to move items within the pathnames and values panes:

- to select several items:
  control+click to add or subtract from the selected group
- to move the selected items:
  re-click and hold on one of the selected items, then drag to the new location

To format an item:

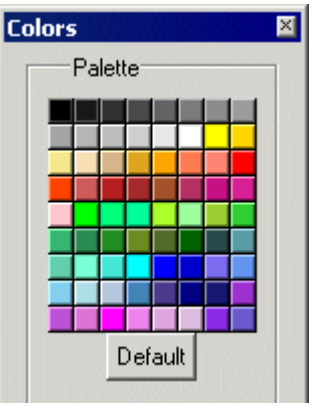
Select the item's label in the pathname pane or its waveform in the waveform pane, then select **View > Signal Properties** (Wave window) or use the selections in the **Format** menu.

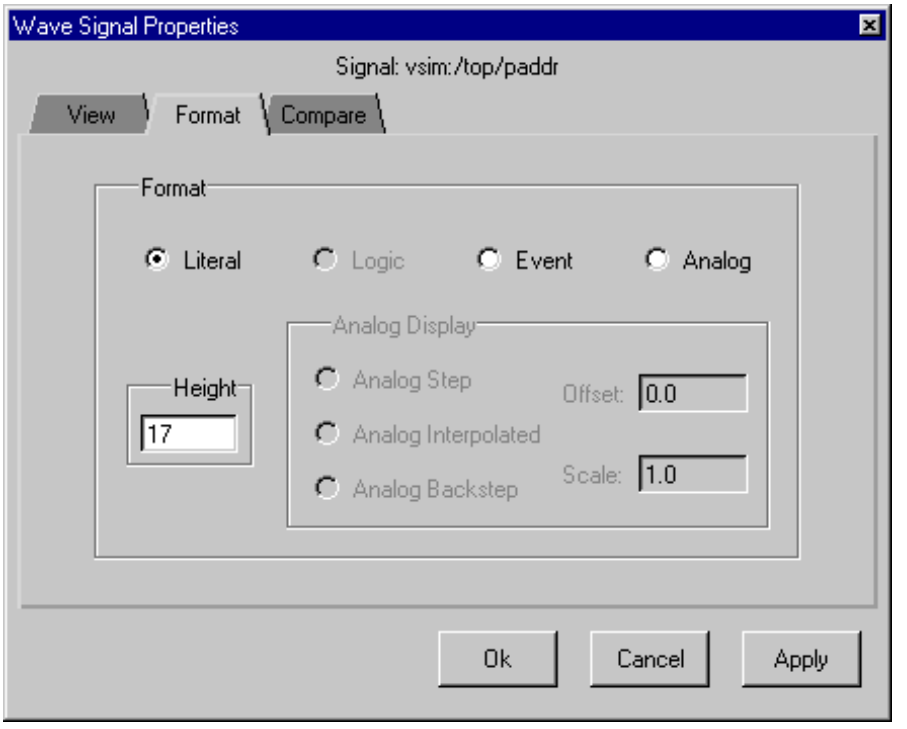
When you select **View > Signal Properties** the Wave Signal Properties dialog box opens. It has three tabs: View, Format, and Compare.



The **View** tab includes these options:

- **Display Name**
  Specifies a new name (in the pathname pane) for the selected signal.
- **Radix**
  Specifies the Radix of the selected signal(s). Setting this to default causes the signal's radix to change whenever the default is modified using the **radix** command (CR-108). Item values are not translated if you select Symbolic.
- **Wave Color**
  Specifies the waveform color. Select a new color from the color palette, or enter a color name. The Default button in the Colors palette allows you to return the selected item's color back to its default value.
- **Name Color**
  Specifies the signal name's color. Select a new color from the color palette, or enter a color name. The Default button in the Colors palette allows you to return the selected item's color back to its default value.





The **Format** tab includes these options (see next page for example graphic):

- **Format: Literal**
  Displays the waveform as a box containing the item value (if the value fits the space available). This is the only format that can be used to list a record.

- **Format: Logic**
  Displays values as U, X, 0, 1, Z, W, L, H, or -.
- **Format: Event**
  Marks each transition during the simulation run.
- **Format: Analog [Step | Interpolated | Backstep]**
  Analog Step
  Displays the waveform in step style.

  Analog Interpolated
  Displays the waveform in interpolated style.

  Analog Backstep
  Displays the waveform in backstep style. Often used for power calculations.

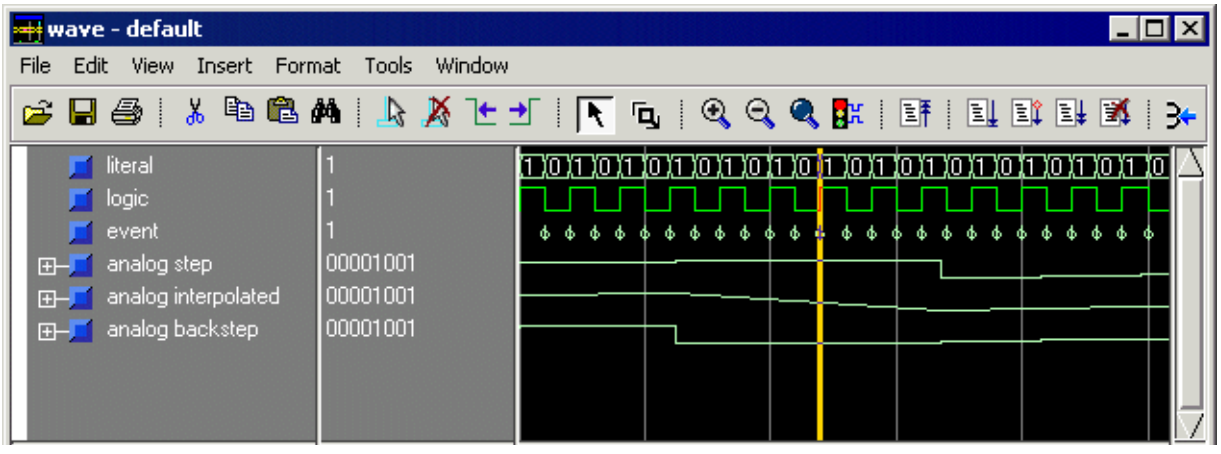
  Offset and Scale
  Allows you to adjust the scale of the item as it is seen on the display. Offset is the number of pixels offset from zero. The scale factor reduces (if less than 1) or increases (if greater than 1) the number of pixels displayed.

  Only the following types are supported in Analog format:

  VHDL types:
  All vectors - std logic vectors, bit vectors, and vectors derived from these types
  Scalar integers
  Scalar reals
  Scalar times

  Verilog types:
  All vectors
  Scalar reals
  Scalar integers
- **Height**
  Allows you to specify the height (in pixels) of the waveform.

The signals in the following illustration demonstrate the various signal formats.

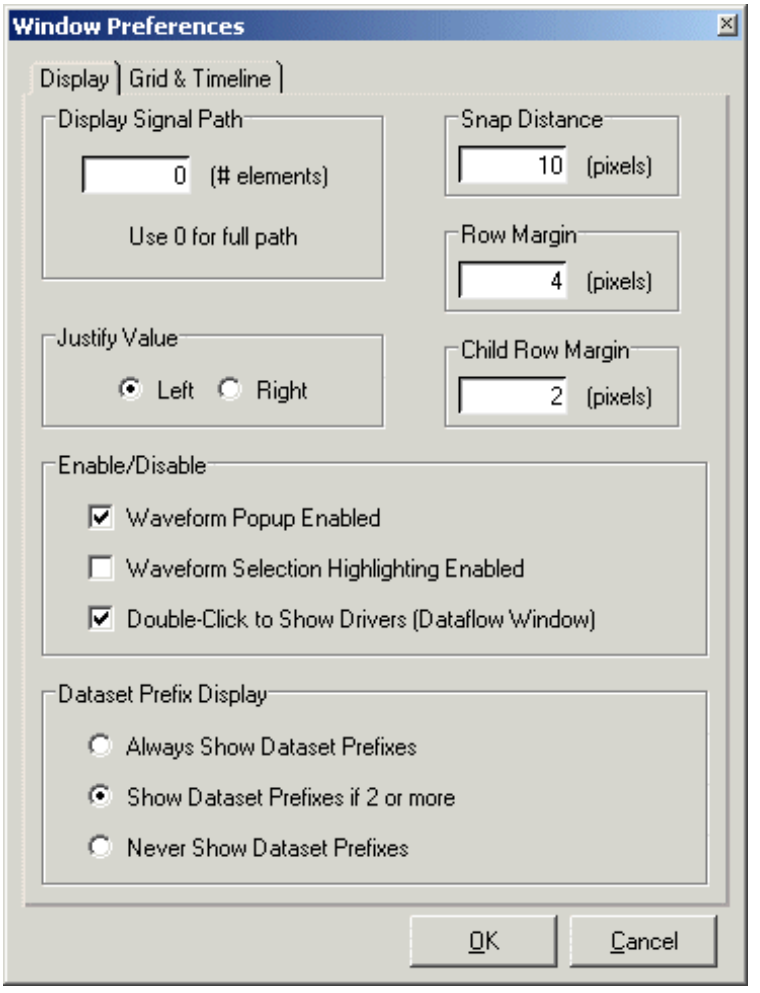


The **Compare** tab includes the same options as those in the Add Signal Options dialog box (see "Comparison Method tab" (UM-309)).

Setting Wave window display properties

You can define display properties of the Wave window by selecting **Tools > Window Preferences** (Wave window). You can make these changes permanent by selecting **Tools > Save Preferences** (Main window). See "Preference variables located in Tcl files" (UM-352) for details on changing window properties permanently.

The dialog box has two tabs–**Display** and **Grid & Timeline**.



The **Display** tab includes the following options:

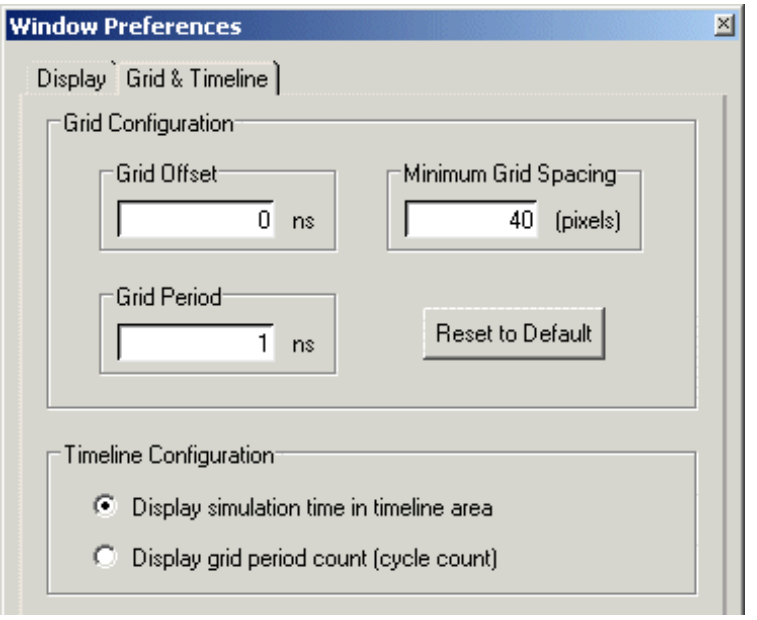
- **Display Signal Path**
  Sets the display to show anything from the full pathname of each signal (e.g., *sim:/top/clk*) to only its leaf element (e.g., *sim:clk*). A non-zero number indicates the number of path elements to be displayed. The default is Full Path.
- **Justify Value**
  Specifies whether the signal values will be justified to the left margin or the right margin in the values window pane.

- **Snap Distance**
  Specifies the distance the cursor needs to be placed from an item edge to jump to that edge (a 0 specification turns off the snap).
- **Row Margin**
  Specifies the distance in pixels between top-level signals.
- **Child Row Margin**
  Specifies the distance in pixels between child signals.
- **Waveform Popup Enable**
  Toggles on/off the popup that displays when you rest your mouse pointer on a signal or comparison object.
- **Waveform Selection Highlighting Enabled**
  Toggles on/off waveform highlighting. When enabled the waveform is highlighted if you select the waveform or its value.
- **Double-Click to Show Drivers (Dataflow Window)**
  Toggles on/off double-clicking to show the drivers of the selected waveform. See "Displaying drivers of the selected waveform" (UM-218) for more details.
- **Dataset Prefix**
  Specifies how signals from different datasets are displayed.

  Always Show Dataset Prefixes
  All dataset prefixes will be displayed along with the dataset prefix of the current simulation ("sim").

  Show Dataset Prefixes if 2 or more
  Displays all dataset prefixes if 2 or more datasets are displayed. "sim" is the default prefix for the current simulation.

  Never Show Dataset Prefixes
  No dataset prefixes will be displayed. This selection is useful if you are running only a single simulation.



The **Grid & Timeline** tab is used to configure grid lines and the horizontal axis in the waveform pane. You can also access this tab by right-clicking in the cursor tracks at the bottom of the Wave window and selecting Grid & Timeline Properties. The tab has the following options:

- **Grid Offset**
  Specifies the time (in user time units) of the first grid line. Default is 0.
- **Grid Period**
  Specifies the time (in user time units) between subsequent grid lines. Default is 1.
- **Minimum Grid Spacing**
  Specifies the closest (in pixels) two grid lines can be drawn before intermediate lines will be removed. Default is 40.
- **Timeline Configuration**
  Specifies whether to display simulation time or grid period count on the horizontal axis. Default is to display simulation time.

Sorting a group of HDL items

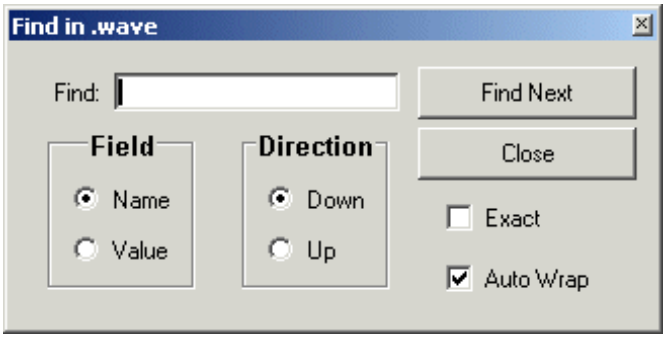
Select **View** > **Sort** to sort the items in the pathname and values panes.

Setting signal breakpoints

You can set "Signal breakpoints" (UM-258) in the Wave window. When a signal breakpoint is hit, a message appears in the Main window Transcript stating which signal caused the breakpoint.

To insert a signal breakpoint, select a signal, click your right mouse button , and select **Insert Breakpoint**. A breakpoint will be set on the selected signal. See "Creating and managing breakpoints" (UM-258) for more information.

Finding items by name or value in the Wave window



The Find dialog box allows you to search for text strings in the Wave window. Select **Edit > Find** (Wave window) to bring up the Find dialog box.

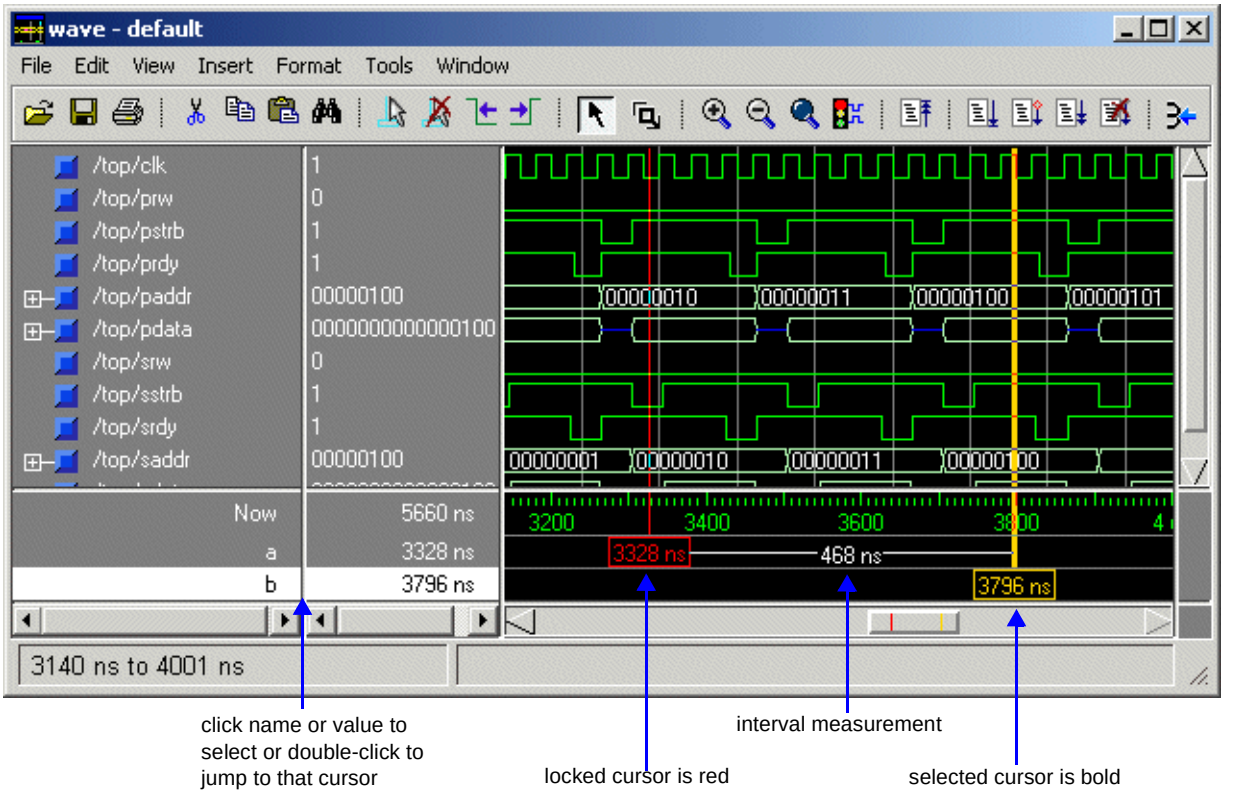
Choose either the Name or Value field to search and enter the value to search for in the Find field. **Find** the item by searching **Down** or **Up** through the Wave window display.

Check **Exact** if you only want to find items that match your search exactly. For example, searching for "clk" without **Exact** will find */top/clk* and *clk1*.

Check **Auto Wrap** to continue the search at the beginning of the window.

The find operation works only within the active pane.

Using time cursors in the Wave window



When the Wave window is first drawn, there is one cursor located at time zero. Clicking anywhere in the waveform display brings that cursor to the mouse location. You can add cursors to the waveform pane by selecting **Insert > Cursor** (or the Add Cursor button shown below). The selected cursor is drawn as a bold solid line; all other cursors are drawn with thin lines. Remove cursors by selecting them and selecting **Edit > Delete Cursor** (or the Delete Cursor button shown below).

Add Cursor add a cursor to the waveform window	**Delete Cursor** delete the selected cursor from the window

Naming cursors

By default cursors are named "Cursor <n>". To rename a cursor, click the name in the left-hand cursor pane with your right mouse button. Type a new name and press the <Enter> key on your keyboard.

Locking cursors

You can lock a cursor in position so it won't move. Click a cursor with your right-mouse button and select **Lock <cursor name>**. The cursor turns red and you can no longer move it with the mouse. As a convenience, you can hold down the <shift> key and click-and-drag the cursor. Once you let go of the cursor, it will be locked in the new position. To unlock a cursor, right-click it and select **Unlock <cursor name>**.

Finding cursors

The cursor value corresponds to the simulation time of that cursor. Choose a specific cursor view by selecting **View > Cursors**.

You can also access cursors by clicking a name or value in the left-hand cursor pane. Single-clicking selects a cursor; double-clicking jumps to a cursor. Alternatively, you can click a value with your second mouse button and type the value to which you want to scroll.

Making cursor measurements

Each cursor is displayed with a time box showing the precise simulation time at the bottom. When you have more than one cursor, each time box appears in a separate track at the bottom of the display. ModelSim also adds a delta measurement showing the time difference between two adjacent cursor positions.

If you click in the waveform display, the cursor closest to the mouse position is selected and then moved to the mouse position. Another way to position multiple cursors is to use the mouse in the time box tracks at the bottom of the display. Clicking anywhere in a track selects that cursor and brings it to the mouse position.

Cursors will "snap" to a waveform edge if you click or drag a cursor to within ten pixels of a waveform edge. You can set the snap distance in the Window Preferences dialog (select **Tools > Window Preferences**). You can position a cursor without snapping by dragging in the cursor track below the waveforms.

You can also move cursors to the next transition of a signal with these toolbar buttons:

Find Previous Transition locate the previous signal value change for the selected signal	**Find Next Transition** locate the next signal value change for the selected signal

Examining waveform values

You can use your mouse to display a dialog that shows the value of a waveform at a particular time. You can do this two ways:

- Rest your mouse pointer on a waveform. After a short delay, a dialog will pop-up that displays the value for the time at which your mouse pointer is positioned. If you'd prefer that this popup not display, it can be toggled off in the display properties. See "Setting Wave window display properties" (UM-222).
- Right-click a waveform and select **Examine**. A dialog displays the value for the time at which you clicked your mouse.

Zooming - changing the waveform display range

Zooming lets you change the simulation range in the waveform pane. You can zoom using the context menu, toolbar buttons, mouse, keyboard, or commands.

You can access Zoom commands from the **View** menu on the toolbar or by clicking the right mouse button in the waveform pane.

The **Zoom** menu options include:

- **Zoom Full**
  Redraws the display to show the entire simulation from time 0 to the current simulation time.
- **Zoom In**
  Zooms in by a factor of two, increasing the resolution and decreasing the visible range horizontally.
- **Zoom Out**
  Zooms out by a factor of two, decreasing the resolution and increasing the visible range horizontally.
- **Zoom Last**
  Restores the display to where it was before the last zoom operation.
- **Zoom Range**
  Brings up a dialog box that allows you to enter the beginning and ending times for a range of time units to be displayed.

Zooming with toolbar buttons

These zoom buttons are available on the toolbar:

Zoom in 2x zoom in by a factor of two from the current view	**Zoom out 2x** zoom out by a factor of two from current view
Zoom Full zoom out to view the full range of the simulation from time 0 to the current time	**Zoom Mode** change mouse pointer to zoom mode; see below

Zooming with the mouse

To zoom with the mouse, first enter zoom mode by selecting **View > Mouse Mode > Zoom Mode** (Wave window). The left mouse button (<Button-1>) then offers 3 zoom options by clicking and dragging in different directions:

- Down-Right *or* Down-Left: Zoom Area (In)
- Up-Right: Zoom Out
- Up-Left: Zoom Fit

The zoom amount is displayed at the mouse cursor. A zoom operation must be more than 10 pixels to activate.

You can also enter zoom mode temporarily by holding the <Ctrl> key down while in select mode.

With the mouse in the Select Mode, the middle mouse button will perform the above zoom operations.

Zooming keyboard shortcuts

See "Wave window mouse and keyboard shortcuts" (UM-231) for a complete list of Wave window keyboard shortcuts.

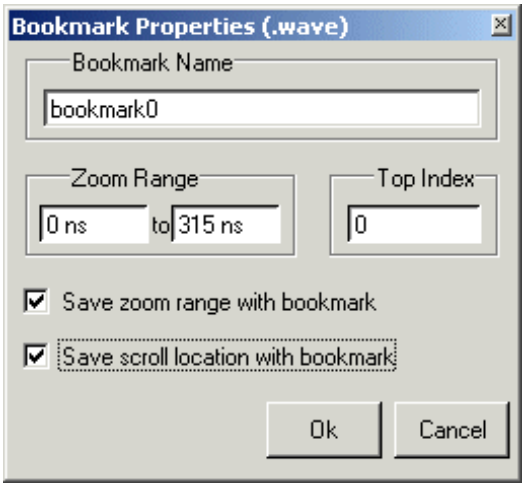
Saving zoom range and scroll position with bookmarks

Bookmarks allow you to save a particular zoom range and scroll position. This lets you return easily to a specific view later. You save the bookmark with a name, and then access the named bookmark from the Bookmark menu.

Bookmarks are saved in the Wave format file (see "Adding items with a Wave window format file" (UM-208)) and are restored when the format file is read. There is no limit to the number of bookmarks you can save.

Bookmarks can also be created and managed from the command line. See the **bookmark add wave** command (CR-42) for details.

To add a bookmark, select **Insert > Bookmark** (Wave window).

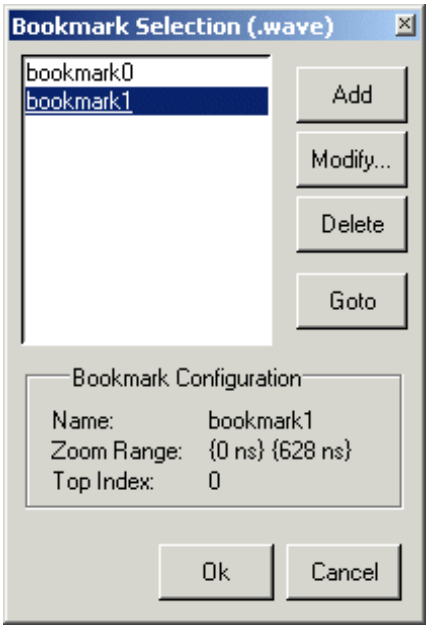


The Bookmark Properties dialog includes the following options.

- **Bookmark Name**
  A text label to assign to the bookmark. The name will identify the bookmark on the View > Bookmarks menu.
- **Zoom Range**
  A starting value and ending value that define the zoom range.
- **Top Index**
  The item that will display at the top of the Wave window. For instance, if you specify 15, the Wave window will be scrolled down to show the 15th item in the window.
- **Save zoom range with bookmark**
  When checked the zoom range will be saved in the bookmark.
- **Save scroll location with bookmark**
  When checked the scroll location will be saved in the bookmark.

Once the bookmark is saved, select it by name from the **View > Bookmarks** menu, and the Wave window will be zoomed and scrolled accordingly.

To edit or delete a bookmark, select **Tools > Bookmarks** (Wave window).



The Bookmark Selection dialog includes the following options.

- **Add** (bookmark add wave)
  Add a new bookmark.
- **Modify**
  Edit the selected bookmark.
- **Delete** (bookmark delete wave)
  Delete the selected bookmark.
- **Goto** (bookmark goto wave)
  Zoom and scroll the Wave window using the selected bookmark.

Wave window mouse and keyboard shortcuts

The following mouse actions and keystrokes can be used in the Wave window.

Mouse action	Result
< control - left-button - drag down and right>[a]	zoom area (in)
< control - left-button - drag up and right>	zoom out
< control - left-button - drag up and left>	zoom fit
<left-button - drag> (Select mode) < middle-button - drag> (Zoom mode)	moves closest cursor
< control - left-button - click on a scroll arrow >	scrolls window to very top or bottom(vertical scroll) or far left or right (horizontal scroll)

a. If you enter zoom mode by selecting **View > Mouse Mode > Zoom Mode**, you do not need to hold down the <Ctrl> key.

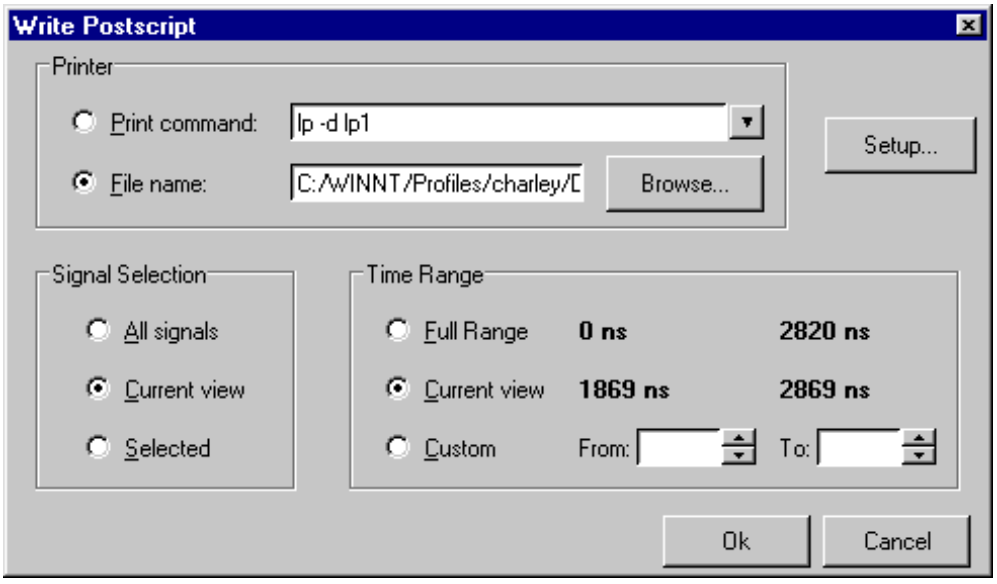
Keystroke	Action
i I or +	zoom in (mouse pointer must be over the the cursor or waveform panes)
o O or -	zoom out (mouse pointer must be over the the cursor or waveform panes)
f or F	zoom full (mouse pointer must be over the the cursor or waveform panes)
l or L	zoom last (mouse pointer must be over the the cursor or waveform panes)
r or R	zoom range (mouse pointer must be over the the cursor or waveform panes)
<up arrow>/ <down arrow>	with mouse over waveform pane, scrolls entire window up/ down one line; with mouse over pathname or values pane, scrolls highlight up/down one line
<left arrow>	scroll pathname, values, or waveform pane left
<right arrow>	scroll pathname, values, or waveform pane right
<page up>	scroll waveform pane up by a page
<page down>	scroll waveform pane down by a page

Keystroke	Action
<tab>	search forward (right) to the next transition on the selected signal - finds the next edge
<shift-tab>	search backward (left) to the previous transition on the selected signal - finds the previous edge
<control-f>	open the find dialog box; searches within the specified field in the pathname pane for text strings
<control-left arrow>	scroll pathname, values, or waveform pane left by a page
<control-right arrow>	scroll pathname, values, or waveform pane right by a page

Saving waveforms

Saving a .eps file

Select **File > Print Postscript** (Wave window) to save the waveform as a .eps file**write wave** command (CR-224). Printing and writing preferences are controlled by the dialog box shown below.



The **Write Postscript** dialog box includes these options:

Printer

- **File name**
  Enter a filename for the encapsulated Postscript (.eps) file to be created; or browse to a previously created .eps file and use that filename.

Signal Selection

- **All signals**
  Print all signals.
- **Current View**
  Print signals in the current view.
- **Selected**
  Print all selected signals.

Time Range

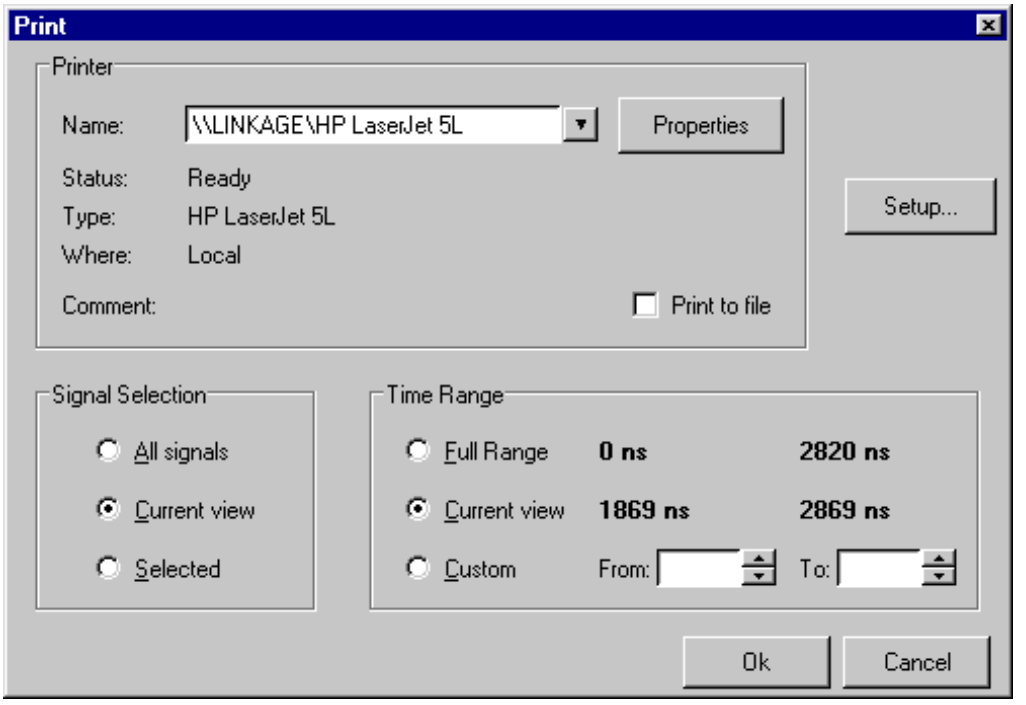
- **Full Range**
  Print all specified signals in the full simulation range.
- **Current view**
  Print the specified signals for the viewable time range.
- **Custom**
  Print the specified signals for a user-designated **From** and **To** time.

Setup button

See "Printer Page Setup" (UM-236)

Printing on Windows platforms

Select **File > Print** (Wave window) to print all or part of the waveform in the current Wave window, or save the waveform as a printer file (a Postscript file for Postscript printers). Printing and writing preferences are controlled by the dialog box shown below.



Printer

- **Name**
  Choose the printer from the drop-down menu. Set printer properties with the **Properties** button.
- **Status**
  Indicates the availability of the selected printer.

- **Type**
  Printer driver name for the selected printer. The driver determines what type of file is output if "Print to file" is selected.
- **Where**
  The printer port for the selected printer.
- **Comment**
  The printer comment from the printer properties dialog box.
- **Print to file**
  Make this selection to print the waveform to a file instead of a printer. The printer driver determines what type of file is created. Postscript printers create a Postscript (.ps) file, non-Postscript printers create a .prn or printer control language file. To create an encapsulated Postscript file (.eps) use the **File > Print Postscript** menu selection.

Signal Selection

- **All signals**
  Print all signals.
- **Current View**
  Print signals in current view.
- **Selected**
  Print all selected signals.

Time Range

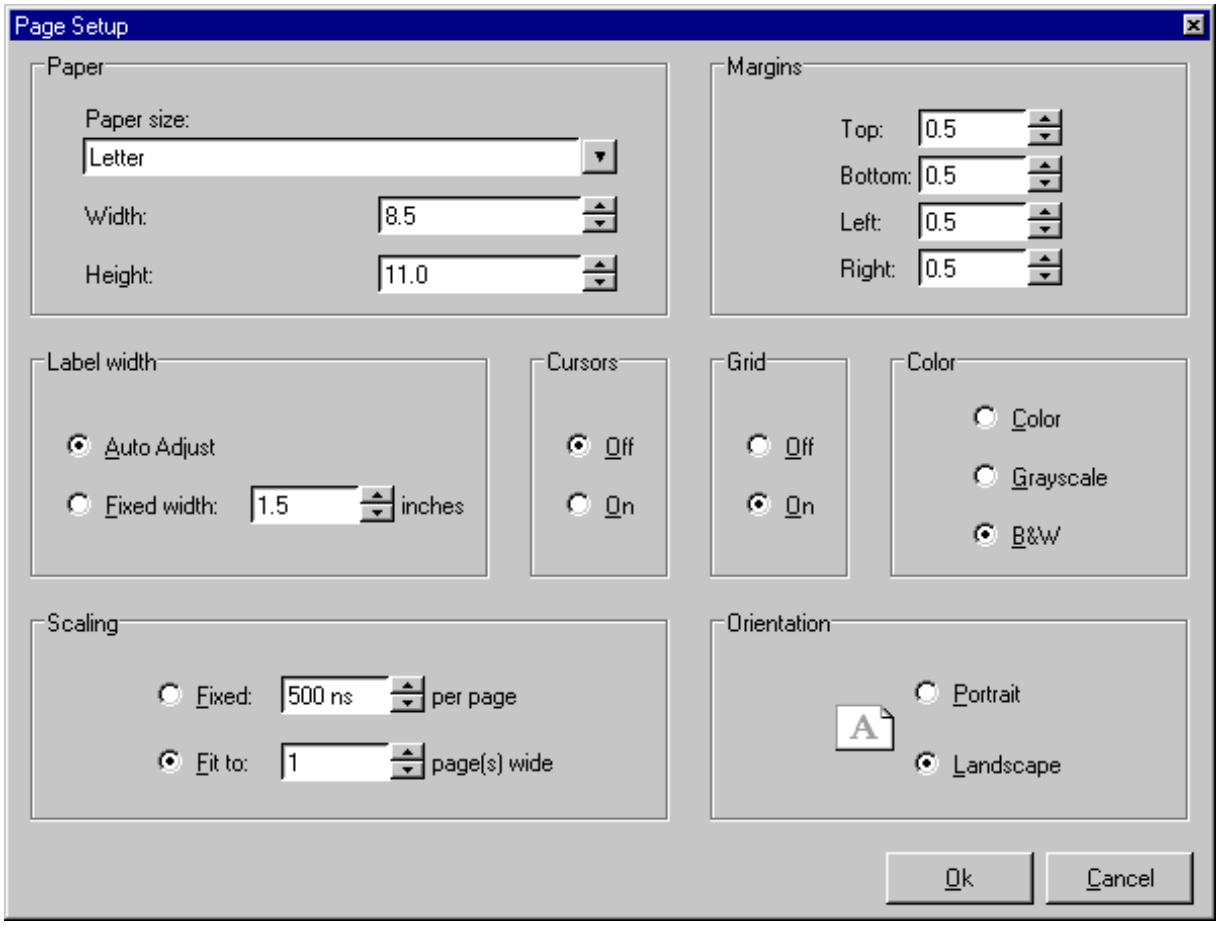
- **Full Range**
  Print all specified signals in the full simulation range.
- **Current view**
  Print the specified signals for the viewable time range.
- **Custom**
  Print the specified signals for a user-designated **From** and **To** time.

Setup button

See "Printer Page Setup" (UM-236)

Printer Page Setup

Clicking the Setup button in the Write Postscript or Print dialog box allows you to define the following options (this is the same dialog that opens via **File > Page setup**).

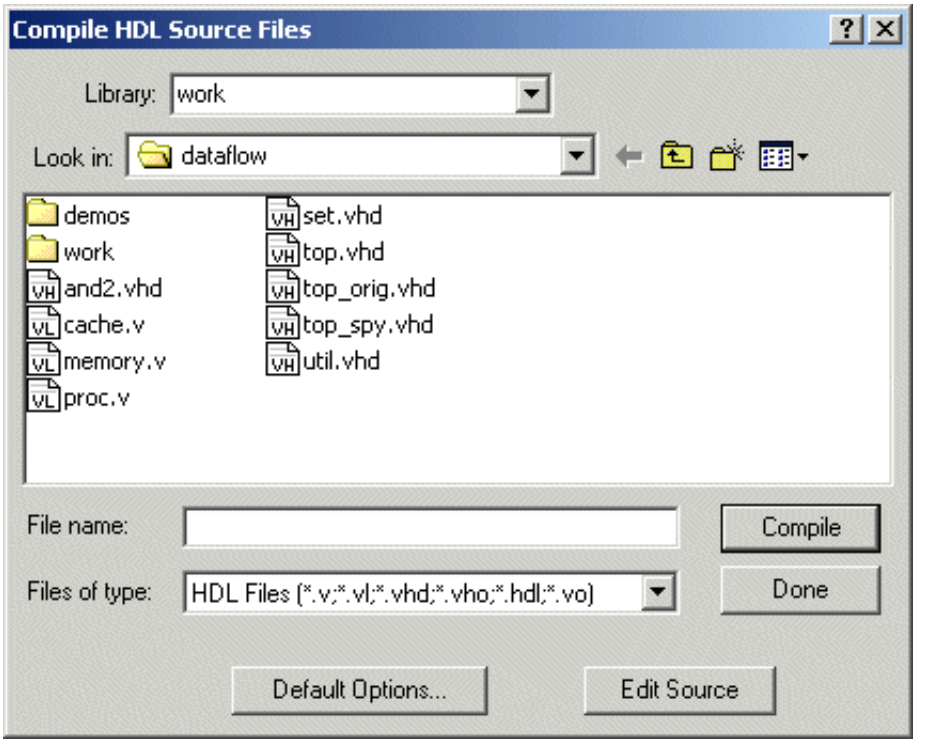


- **Paper Size**
  Select your output page size from a number of options; also choose the paper width and height.
- **Margins**
  Specify the page margins; changing the **Margin** will change the **Scale** and **Page** specifications.
- **Label width**
  Specify Auto Adjust to accommodate any length label, or set a fixed label width.
- **Cursors**
  Turn printing of cursors on or off.
- **Grid**
  Turn printing of grid lines on or off.

- **Color**
  Select full color printing, grayscale, or black and white.
- **Scaling**
  Specify a **Fixed** output time width in nanoseconds per page – the number of pages output is automatically computed; or, select **Fit to** to define the number of pages to be output based on the paper size and time settings; if set, the time-width per page is automatically computed.
- **Orientation**
  Select the output page orientation, **Portrait** or **Landscape**.

Compiling with the graphic interface

You can use a project or the **Compile HDL Source Files** dialog box to compile VHDL or Verilog designs. For information on compiling in a project, see "Getting started with projects" (UM-20). To open the Compile HDL Source Files dialog, select **Compile > Compile** (Main window).



From the Compile HDL Source Files dialog box you can:

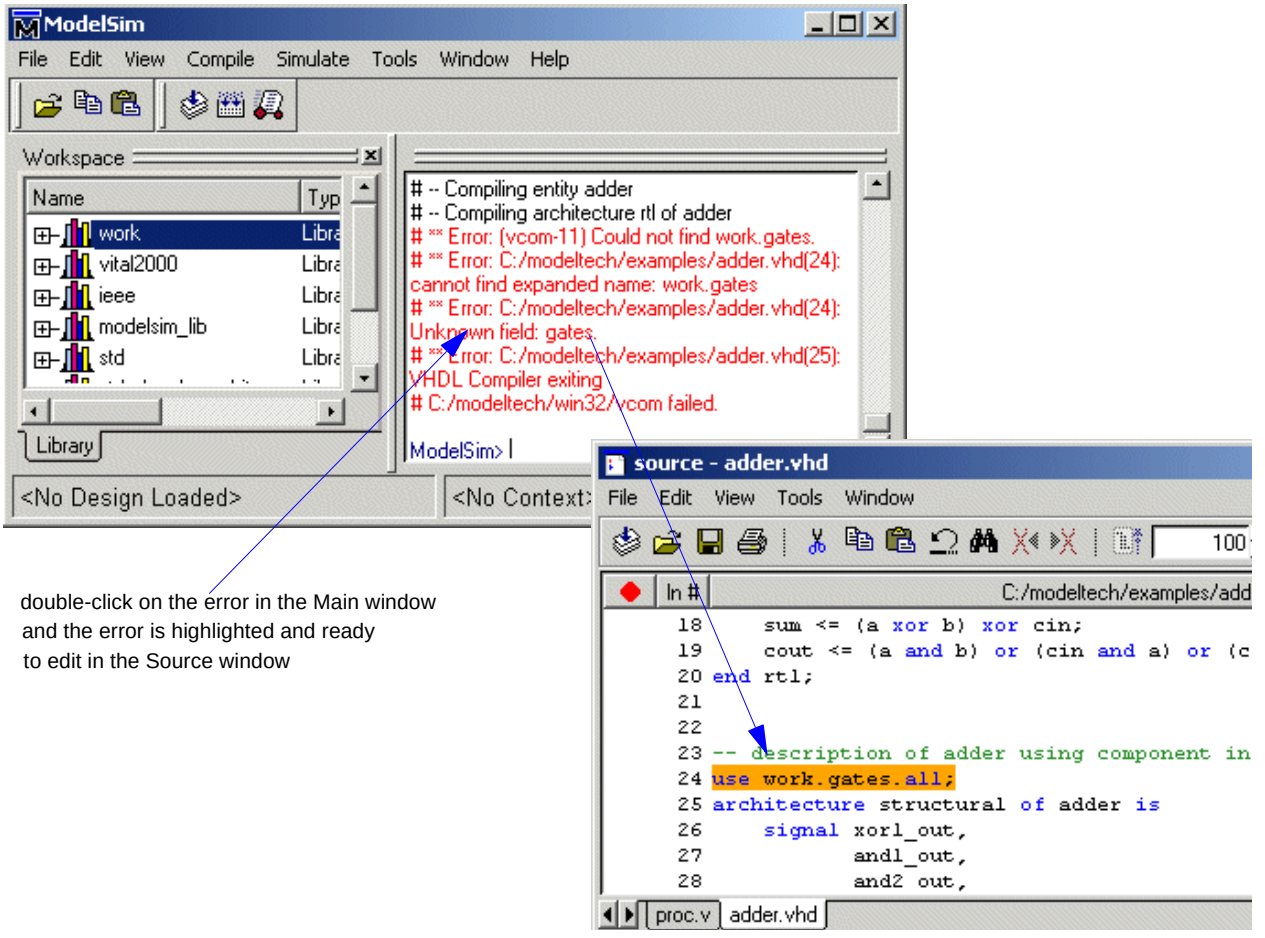
- select source files to compile in any language combination
- specify the target library for the compiled design units
- select among the compiler options for either VHDL or Verilog

Select the **Default Options** button to change the compiler options, see "Setting default compile options" (UM-240) for details. The same Compiler Options dialog box can also be accessed by selecting **Compile > Compile Options** (Main window) or by selecting Compile Properties from the context menu in the Project tab.

Select the **Edit Source** button to view or edit a source file via the Compile dialog box. See "Source window" (UM-191) for additional source file editing information.

Locating source errors during compilation

If a compiler error occurs during compilation, a red error message is printed in the Main transcript. Double-click on the error message to open the source file in an editable Source window with the error highlighted.

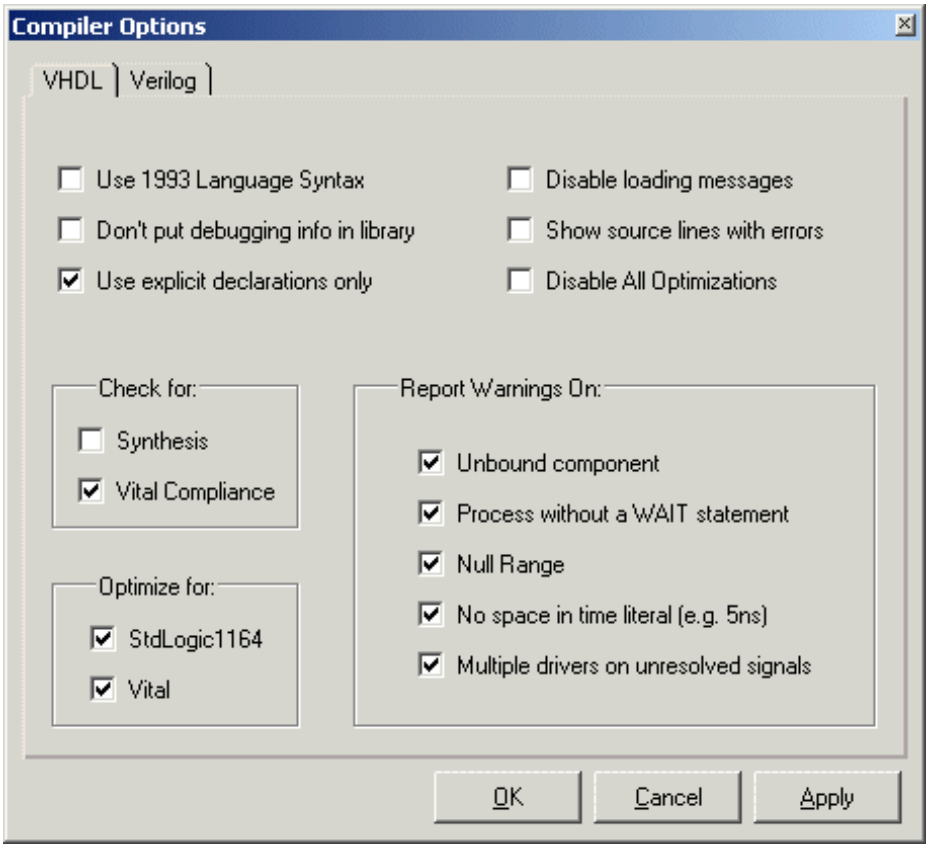


Setting default compile options

Select **Compile** > **Compile Options** (Main window) to bring up the Compiler Options dialog.

Important: Note that changes made in the **Compiler Options** dialog box become the default for all future simulations.

VHDL compiler options tab



The VHDL compiler options tab includes the following options:

- **Use 1993 Language Syntax**
  Specifies the use of VHDL93 during compilation. The 1987 standard is the default. Same as the **-93** argument to the **vcom** command (CR-145). Edit the VHDL93 (UM-351) variable in the *modelsim.ini* file to set a permanent default.

- **Use explicit declarations only**
  Used to ignore an error in packages supplied by some other EDA vendors; directs the compiler to resolve ambiguous function overloading in favor of the explicit function definition. Same as the **-explicit** argument to the **vcom** command (CR-145). Edit the Explicit (UM-342) variable in the *modelsim.ini* file to set a permanent default.

  Although it is not intuitively obvious, the = operator is overloaded in the **std_logic_1164** package. All enumeration data types in VHDL get an "implicit" definition for the = operator. So while there is no explicit = operator, there is an implicit one. This implicit declaration can be hidden by an explicit declaration of = in the same package (LRM Section 10.3). However, if another version of the = operator is declared in a different package than that containing the enumeration declaration, and both operators become visible through **use** clauses, neither can be used without explicit naming, for example:

  ```
  ARITHMETIC."="(left, right)
  ```

  This option allows the explicit = operator to hide the implicit one.

- **Disable loading messages**
  Disables loading messages in the Main window. Same as the **-quiet** argument for the **vcom** command (CR-145). Edit the Quiet (UM-342) variable in the *modelsim.ini* file to set a permanent default.

- **Show source lines with errors**
  Causes the compiler to display the relevant lines of code in the transcript. Same as the **-source** argument to the **vcom** command (CR-145). Edit the Show_source (UM-342) variable in the *modelsim.ini* file to set a permanent default.

- **Disable All Optimizations**
  Instructs the compiler to remove all optimizations. Same as the **-O0** argument to the **vcom** command (CR-145). Useful when running "Code Coverage" (UM-283), where optimizations can skew results.

Flag Warnings on:

- **Unbound Component**
  Flags any component instantiation in the VHDL source code that has no matching entity in a library that is referenced in the source code, either directly or indirectly. Edit the Show_Warning1 (UM-343) variable in the *modelsim.ini* file to set a permanent default.

- **Process without a WAIT statement**
  Flags any process that does not contain a wait statement or a sensitivity list. Edit the Show_Warning2 (UM-343) variable in the *modelsim.ini* file to set a permanent default.

- **Null Range**
  Flags any null range, such as 0 down to 4. Edit the Show_Warning3 (UM-343) variable in the *modelsim.ini* file to set a permanent default.

- **No space in time literal (e.g. 5ns)**
  Flags any time literal that is missing a space between the number and the time unit. Edit the Show_Warning4 (UM-343) variable in the *modelsim.ini* file to set a permanent default.

- **Multiple drivers on unresolved signals**
  Flags any unresolved signals that have multiple drivers. Edit the Show_Warning5 (UM-343) variable in the *modelsim.ini* file to set a permanent default.

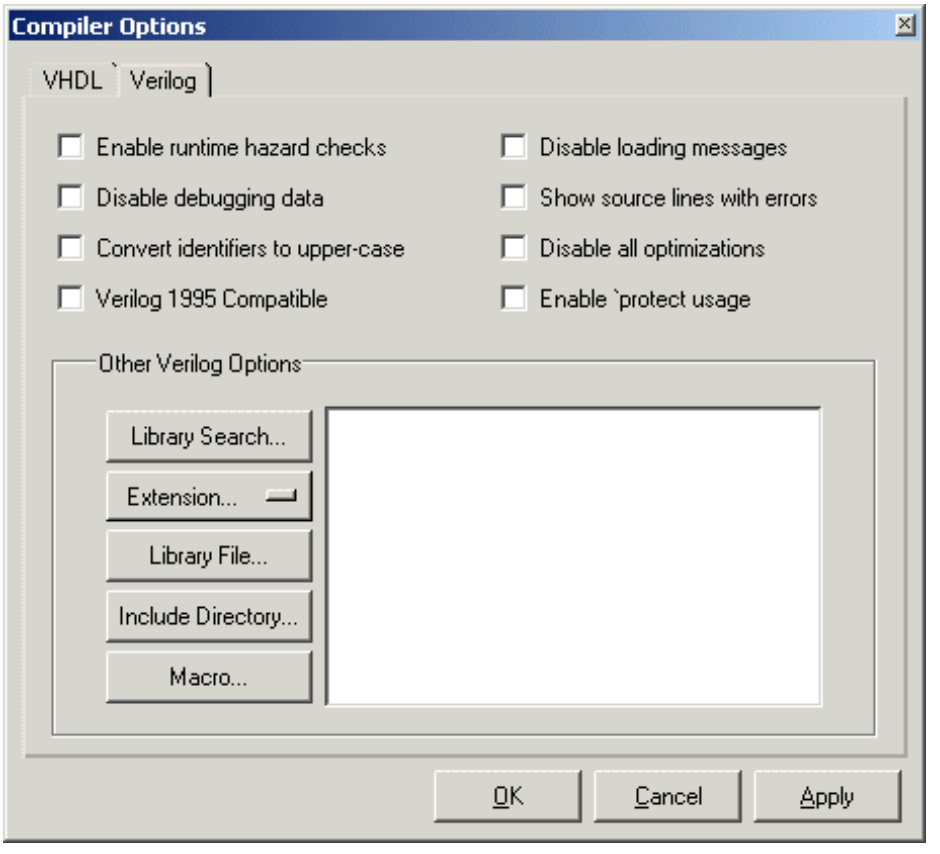
Check for:

- **Synthesis**
  Turns on limited synthesis-rule compliance checking. Checks only signals used (read) by a process; also, checks understand only combinational logic, not clocked logic. Edit the CheckSynthesis (UM-342) variable in the *modelsim.ini* file to set a permanent default.
- **Vital Compliance**
  Toggle Vital compliance checking. Edit the NoVitalCheck (UM-342) variable in the *modelsim.ini* file to set a permanent default.

Optimize for:

- **StdLogic1164**
  Causes the compiler to perform special optimizations for speeding up simulation when the multi-value logic package std_logic_1164 is used. Unless you have modified the std_logic_1164 package, this option should always be checked. Edit the Optimize_1164 (UM-342) variable in the *modelsim.ini* file to set a permanent default.
- **Vital**
  Toggle acceleration of the Vital packages. Edit the NoVital (UM-342) variable in the *modelsim.ini* file to set a permanent default.

Verilog compiler options tab



- **Enable runtime hazard checks**
  Enables the run-time hazard checking code. Same as the **-hazards** argument to the **vlog** command (CR-181). Edit the Hazard (UM-343) variable in the *modelsim.ini* file to set a permanent default.
- **Convert identifiers to upper-case**
  Converts regular Verilog identifiers to uppercase. Allows case insensitivity for module names. Same as the **-u** argument to the **vlog** command (CR-181). Edit the UpCase (UM-343) variable in the *modelsim.ini* file to set a permanent default.
- **Verilog 1995 Compatible**
  Some requirements in Verilog 2000 conflict with requirements in the 1995 LRM. Use of this option ensures that code that was valid according to the 1995 LRM can still be compiled. Same as the **-vlog59compat** argument for the **vlog** command (CR-181). Edit the vlog95compat (UM-343) variable in the *modelsim.ini* file to set a permanent default.
- **Disable loading messages**
  Disables loading messages in the Main window. Same as the **-quiet** argument for the **vlog** command (CR-181). Edit the Quiet (UM-342) variable in the *modelsim.ini* file to set a permanent default.
- **Show source lines with errors**
  Causes the compiler to display the relevant lines of code in the transcript. Same as the **-source** argument to the **vlog** command (CR-181). Edit the Show_source (UM-342) variable in the *modelsim.ini* file to set a permanent default.
- **Disable All Optimizations**
  Instructs the compiler to remove all optimizations. Same as the **-O0** argument to the **vlog** command (CR-181). Useful when running "Code Coverage" (UM-283), where optimizations can skew results.

Other Verilog Options:

- **Library Search**
  Specifies the Verilog source library directory to search for undefined modules. Same as the **-y <library_directory>** argument for the **vlog** command (CR-181).
- **Extension**
  Specifies the suffix of files in the library directory. Multiple suffixes can be used. Same as the **+libext+<suffix>** argument for the **vlog** command (CR-181).
- **Library File**
  Specifies the Verilog source library file to search for undefined modules. Same as the **-v <library_file>** argument for the **vlog** command (CR-181).
- **Include Directory**
  Specifies a directory for files included with the **‘include filename** compiler directive. Same as the **+incdir+<directory>** argument for the **vlog** command (CR-181).
- **Macro**
  Defines a macro to execute during compilation. Same as the compiler directive: ‘define

macro_name macro_text. Also the same as the **+define+<macro_name> [=<macro_text>]** argument for the **vlog** command (CR-181).

Note: When you specify Other Verilog Options, they are saved into a file called *vlog.opt*. If you do this while a project is open, an OptionFile entry is written into your project file. If you do this when a project is not open, an OptionFile entry is written into the *modelsim.ini* file that you are currently using.

Simulating with the graphic interface

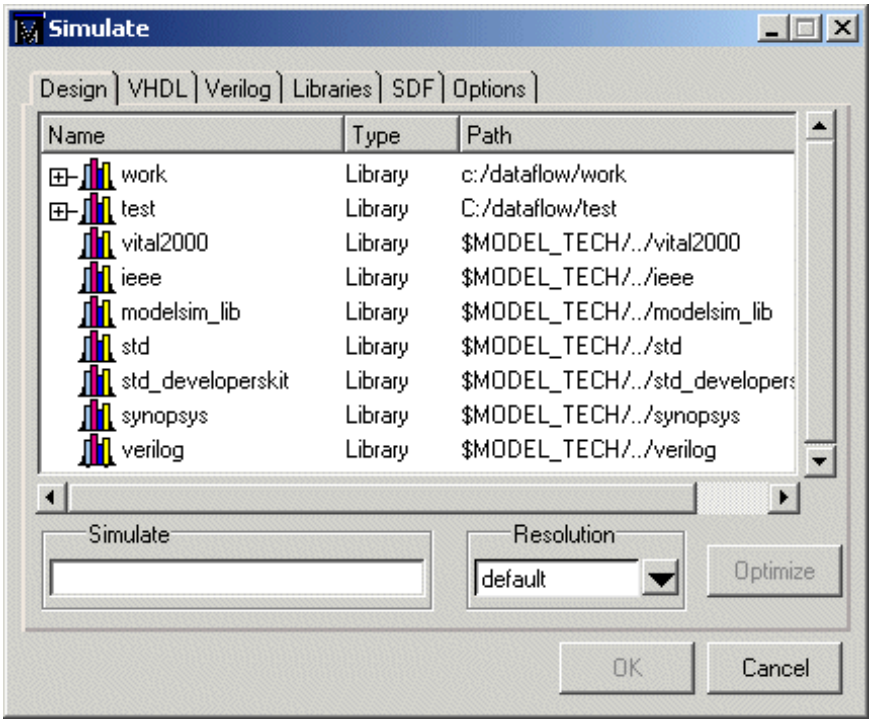
You can use the Library tab in the workspace or the **Simulate** dialog box to simulate a compiled design. To simulate from the Library tab, simply double-click a design unit. To open the **Simulate** dialog, select **Simulate > Simulate** (Main window).

Six tabs - **Design**, **VHDL**, **Verilog**, **Libraries**, **SDF**, and **Options** - allow you to select various simulation options.

You can switch between tabs to modify settings, then begin simulation by selecting the **OK** button.

Note: To begin simulation you must have compiled design units located in a design library, see "Creating a design library" (UM-50).

Design tab



The **Design** tab includes these options:

- **Simulate**
  Specifies the design unit(s) to simulate. You can simulate several Verilog top-level modules or a VHDL top-level design unit in one of three ways:

  - Type a design unit name (configuration, module, or entity) into the field, separate additional names with a space. Specify library/design units with the following syntax:

    ```
    [<library_name>.]<design_unit>
    ```

  - Select a design unit from the list. You can select multiple design units from the list by using the control key when you click.

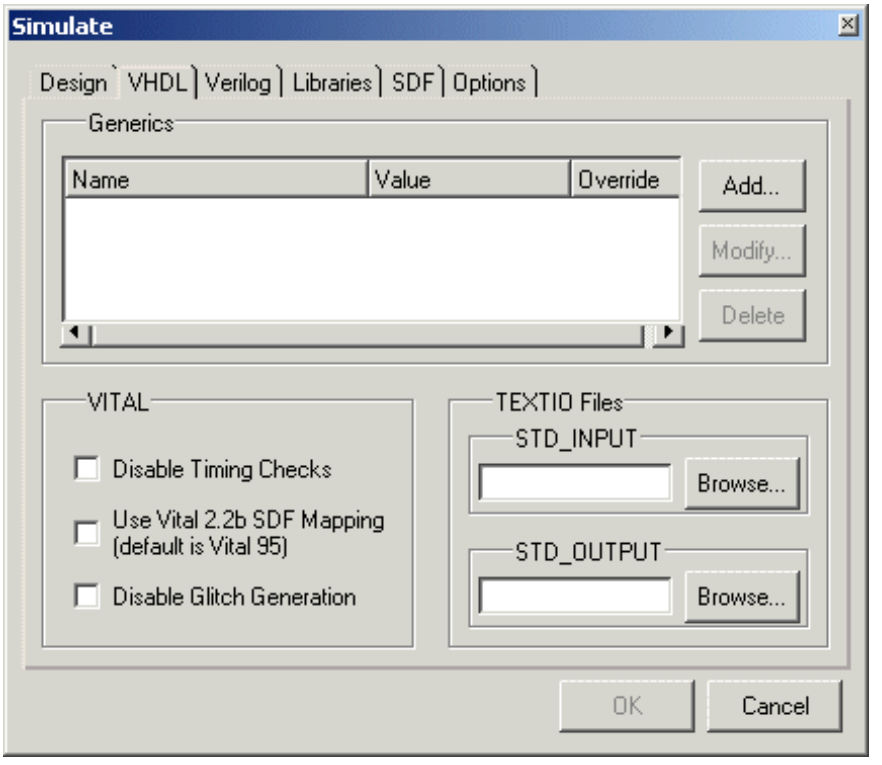
- **Resolution**
  (-t [<multiplier>]<time_unit>)
  The drop-down menu sets the simulator time units.

  Simulator time units can be expressed as any of the following:

Simulation time units	
1fs, 10fs, or 100fs	femtoseconds
1ps, 10ps, or 100ps	picoseconds
1ns, 10ns, or 100ns	nanoseconds
1us, 10us, or 100us	microseconds
1ms, 10ms, or 100ms	milliseconds
1sec, 10sec, or 100sec	seconds

See also, "Simulator resolution limit" (UM-52).

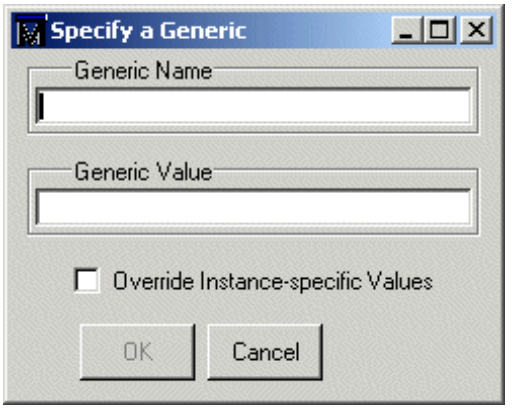
VHDL tab



The **VHDL** tab includes these options:

Generics

The **Add** button opens a dialog box (shown below) that allows you to specify the value of generics within the current simulation; generics are then added to the **Generics** list. You can also select a generic on the listing to **Delete** or **Edit**.



From the **Specify a Generic** dialog box you can set the following options.

- **Generic Name** (-g <Name>=<Value>)
  The name of the generic parameter. Type it in as it appears in the VHDL source (case is ignored).
- **Generic Value**
  Specifies a value for all generics in the design with the given name (above) that have not received explicit values in generic maps (such as top-level generics and generics that

would otherwise receive their default value). The value must be appropriate for the declared data type of the generic. No spaces are allowed in the specification (except within quotes) when specifying a string value.

- **Override Instance - specific Values** (-G <Name>=<Value>)
  Select to override generics that received explicit values in generic maps. The name and value are specified as above. The use of this switch is indicated in the **Override** column of the **Generics** list.

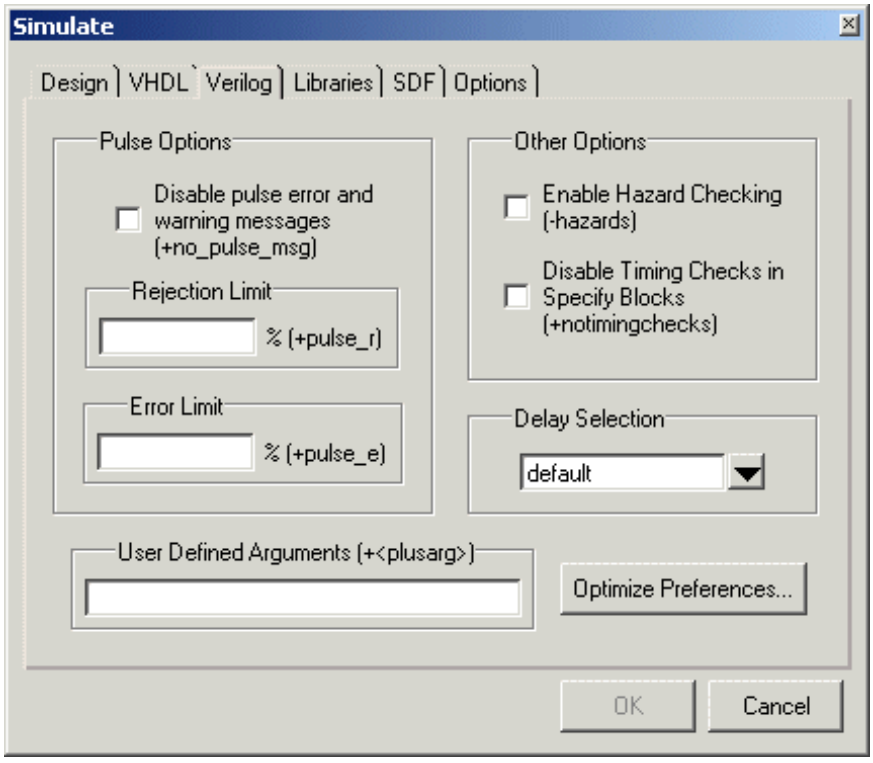
VITAL

- **Disable Timing Checks** (+notimingchecks)
  Disables timing checks generated by VITAL models.
- **Use Vital 2.2b SDF Mapping** (-vital2.2b)
  Selects SDF mapping for VITAL 2.2b (default is Vital95).
- **Disable Glitch Generation** (-noglitch)
  Disables VITAL glitch generation.

TEXTIO files

- **STD_INPUT** (-std_input <filename>)
  Specifies the file to use for the VHDL textio STD_INPUT file. Use the **Browse** button to locate a file within your directories.
- **STD_OUTPUT** (-std_output <filename>)
  Specifies the file to use for the VHDL textio STD_OUTPUT file. Use the **Browse** button to locate a file within your directories.

Verilog tab



The **Verilog** tab includes these options:

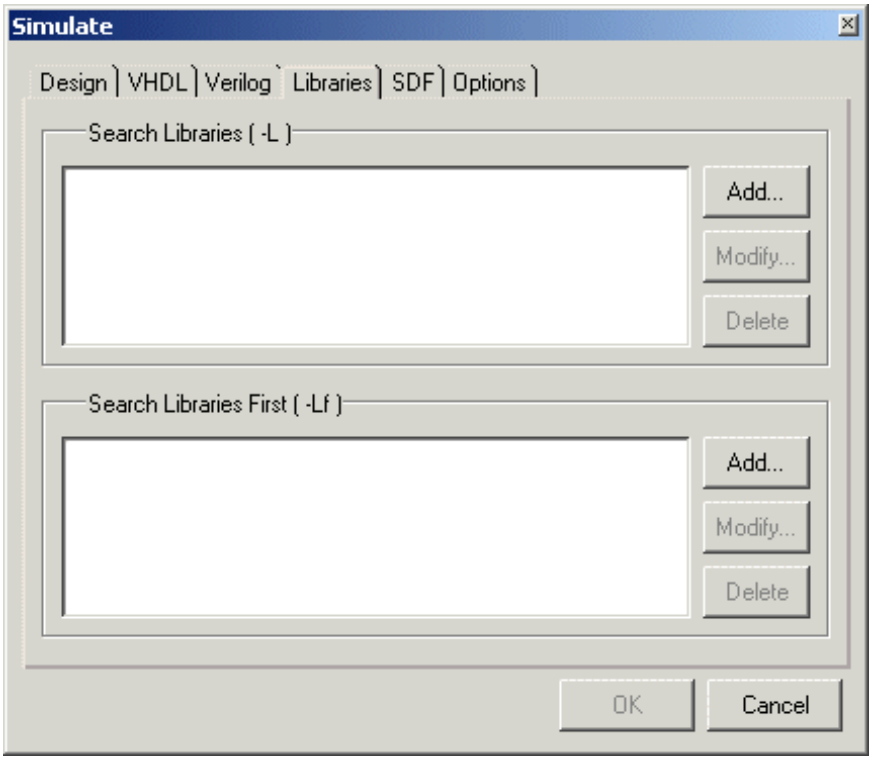
Pulse Options

- **Disable pulse error and warning messages** (+no_pulse_msg)
  Disables path pulse error warning messages.
- **Rejection Limit** (+pulse_r/<percent>)
  Sets the module path pulse rejection limit as a percentage of the path delay.
- **Error Limit** (+pulse_e/<percent>)
  Sets the module path pulse error limit as a percentage of the path delay.

Other Options

- **Enable Hazard Checking** (-hazards)
  Enables hazard checking in Verilog modules.
- **Disable Timing Checks in Specify Blocks** (+notimingchecks)
  Disables the timing check system tasks ($setup, $hold,...) in specify blocks.
- **Delay Selection** (+mindelays | +typdelays | +maxdelays)
  Use the drop-down menu to select timing for min:typ:max expressions.
- **User Defined Arguments** (+<plusarg>)
  Arguments are preceded with “+”, making them accessible through the Verilog PLI routine **mc_scan_plusargs**. The values specified in this field must have a "+" preceding them or ModelSim may parse them incorrectly.

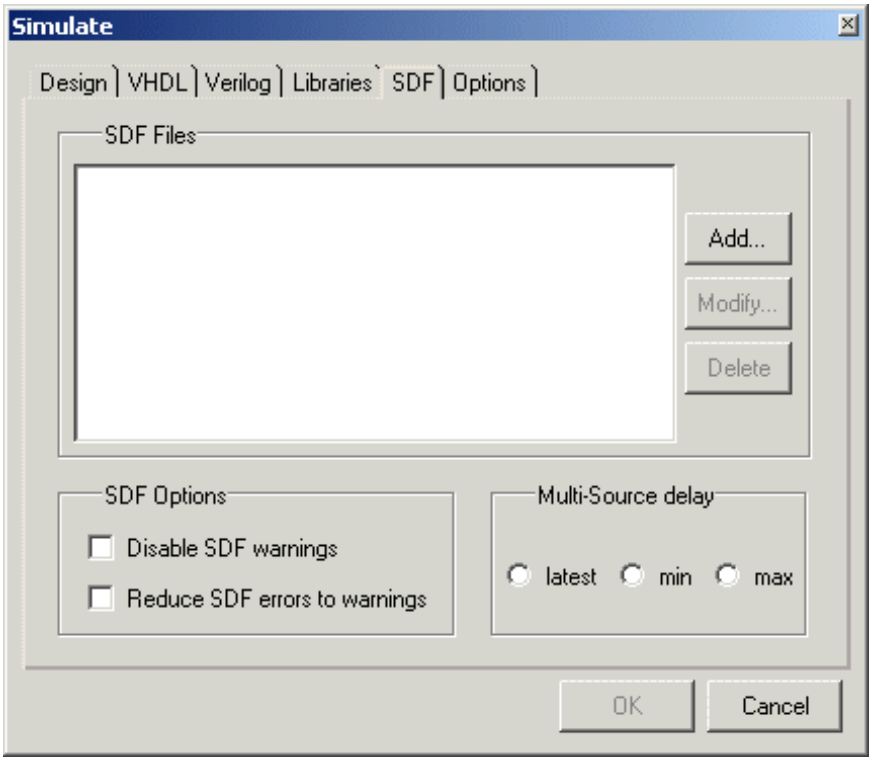
Libraries tab



The **Libraries** tab includes these options:

- **Search Libraries** (-L)
  Specifies the libraries to search for design units instantiated from Verilog.
- **Search Libraries First** (-Lf)
  Same as Search Libraries but these libraries are searched before ‘uselib.

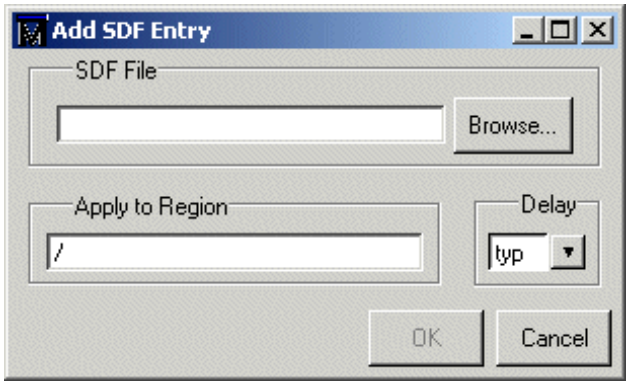
SDF tab



The **SDF** (Standard Delay Format) tab includes these options:

SDF Files

Click the **Add** button to specify the SDF files to load for the current simulation; files are then added to the **Region/File** list. You may also select a file on the listing to **Delete** or **Edit** (opens the dialog box below).



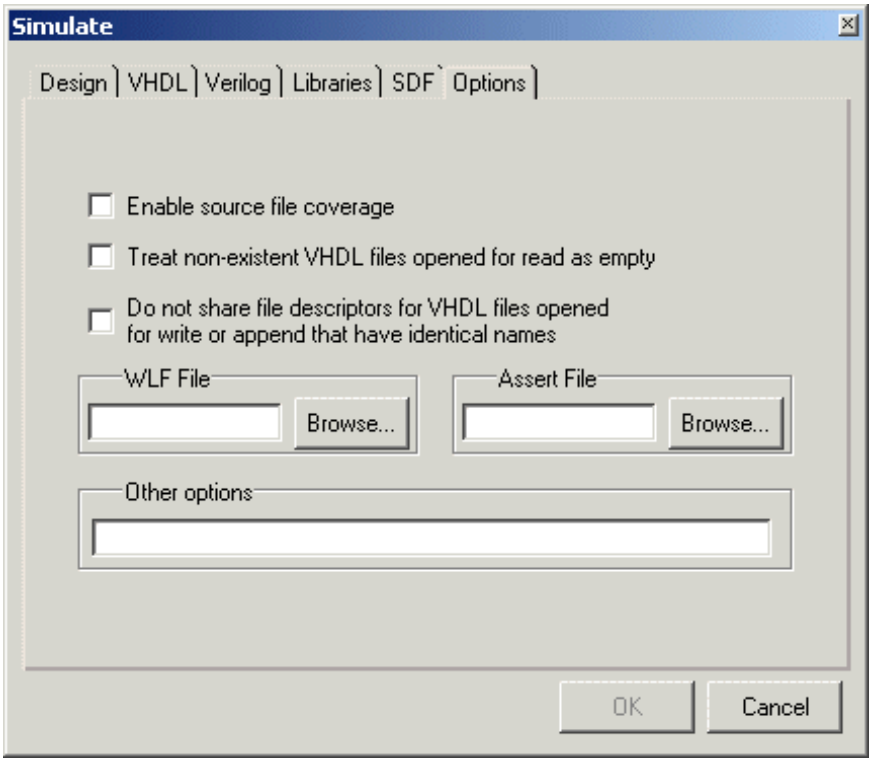
From the **Add SDF File** dialog box you can set the following options.

- **SDF file** ([<region>] = <sdf_filename>)
  Specifies the SDF file to use for annotation. Use the **Browse** button to locate a file within your directories.
- **Apply to region** ([<region>] = <sdf_filename>)
  Specifies the design region to use with the selected SDF options.
- **Delay** (-sdfmin | -sdftyp | -sdfmax)
  The drop-down menu selects delay timing (min, typ or max) to be used from the specified SDF file. See also, "Specifying SDF files for simulation" (UM-290).

SDF options

- **Disable SDF warnings** (-sdfnowarn)
  Select to disable warnings from the SDF reader.
- **Reduce SDF errors to warnings** (-sdfnoerror)
  Change SDF errors to warnings so the simulation can continue.
- **Multi-Source Delay** (-multisource_delay <sdf_option>)
  Select **max**, **min** or **latest** delay. Controls how multiple PORT or INTERCONNECT constructs that terminate at the same port are handled. By default, the Module Input Port Delay (MIPD) is set to the **max** value encountered in the SDF file. Alternatively, you can choose the **min** or **latest** of the values.

Options tab



The **Options** tab includes these options:

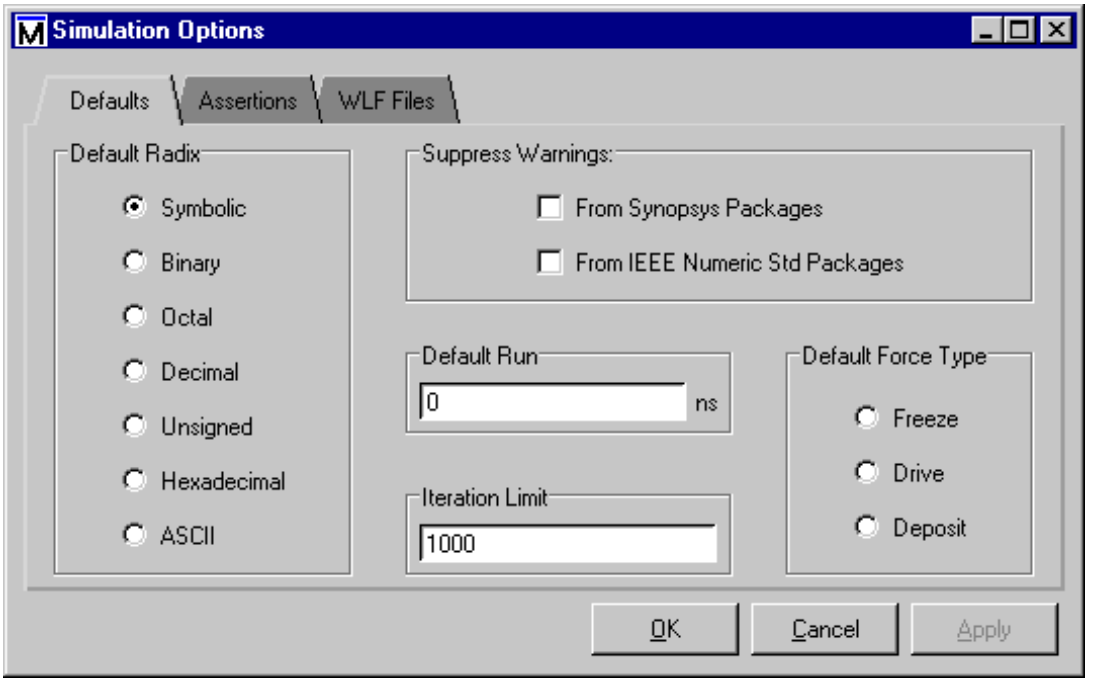
- **Enable source file coverage** (-coverage)
  Turn on collection of Code Coverage statistics. See *Chapter 9 - Code Coverage*.
- **Treat non-existent VHDL files ...** (-absentisempty)
  Cause VHDL files opened for read that target non-existent files to be treated as empty, rather than ModelSim issuing fatal error messages.
- **Do not share file descriptors...** (-nofileshare)
  By default ModelSim shares a file descriptor for all VHDL files opened for write or append that have identical names. This option turns off file descriptor sharing.
- **WLF File** (-wlf <filename>)
  Specify the name of the wave log format (WLF) file to create. The default is vsim.wlf.
- **Assert File** (-assertfile <filename>)
  Designate an alternative file for recording assertion messages. By default assertion messages are output to the file specified by the TranscriptFile variable in the *modelsim.ini* file (see "Creating a transcript file" (UM-349)).
- **Other options**
  Specify any other **vsim** command (CR-189) arguments.

Setting default simulation options

Select **Simulate > Simulation Options** (Main window) to bring up the **Simulation Options** dialog box shown below.

Note: Changes made in the **Simulation Options** dialog box are the default for the current simulation only. Options can be saved as the default for future simulations by editing the simulator control variables in the *modelsim.ini* file; the variables to edit are noted in the text below.

Defaults tab



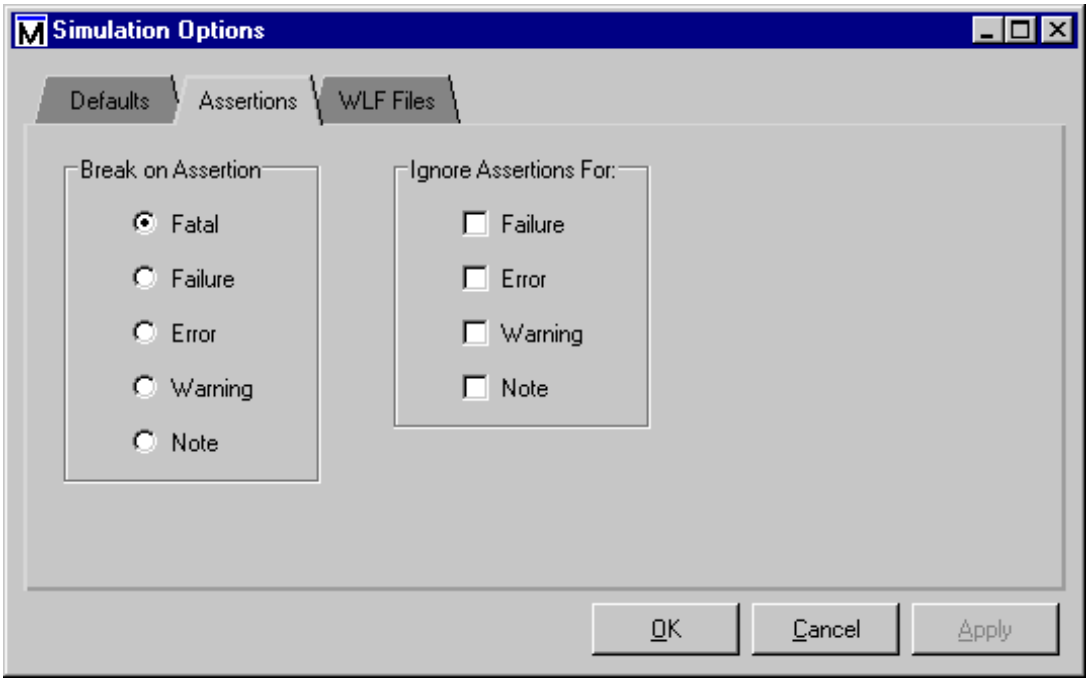
The **Defaults** tab includes these options:

- **Default Radix**
  Sets the default radix for the current simulation run. You can also use the **radix** (CR-108) command to set the same temporary default. A permanent default can be set by editing the DefaultRadix (UM-345) variable in the *modelsim.ini* file. The chosen radix is used for all commands (**force** (CR-82), **examine** (CR-75), **change** (CR-50) are examples) and for displayed values in the Signals, Variables, Dataflow, List, and Wave windows.
- **Suppress Warnings**
  Selecting **From Synopsys Packages** suppresses warnings generated within the accelerated Synopsys std_arith packages. Edit the StdArithNoWarnings (UM-347) variable in the *modelsim.ini* file to set a permanent default.

  Selecting **From IEEE Numeric Std Packages** suppresses warnings generated within the accelerated numeric_std and numeric_bit packages. Edit the NumericStdNoWarnings (UM-347) variable in the *modelsim.ini* file to set a permanent default.

- **Default Run**
  Sets the default run length for the current simulation. Edit the RunLength (UM-347) variable in the *modelsim.ini* file to set a permanent default.
- **Iteration Limit**
  Sets a limit on the number of deltas within the same simulation time unit to prevent infinite looping. Edit the IterationLimit (UM-346) variable in the *modelsim.ini* file to set a permanent iteration limit default.
- **Default Force Type**
  Selects the default force type for the current simulation. Edit the DefaultForceKind (UM-345) variable in the *modelsim.ini* file to set a permanent default.

Assertions tab



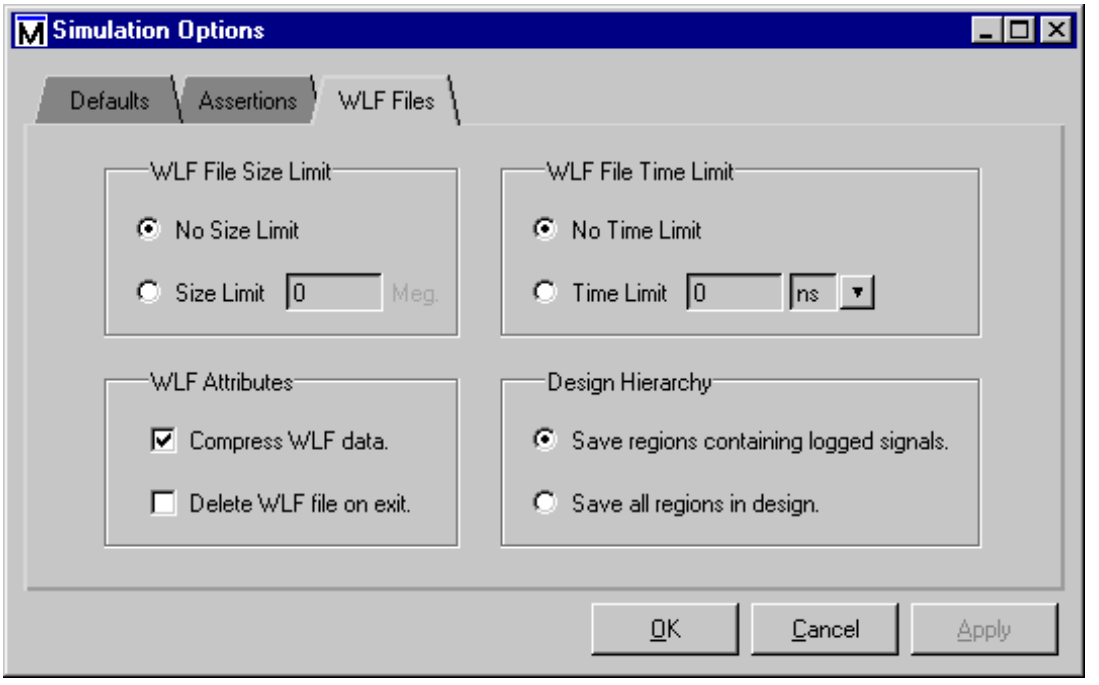
The **Assertions** tab includes these options:

- **Break on Assertion**
  Selects the assertion severity that will stop simulation. Edit the BreakOnAssertion (UM-345) variable in the *modelsim.ini* file to set a permanent default.
- **Ignore Assertions For**
  Selects the assertion type to ignore for the current simulation. Multiple selections are possible. Edit the IgnoreFailure, IgnoreError, IgnoreWarning, and IgnoreNote (UM-346) variables in the *modelsim.ini* file to set permanent defaults.

When an assertion type is ignored, no message will be printed, nor will the simulation halt (even if break on assertion is set for that type).

Note: Assertions that appear within an instantiation or configuration port map clause conversion function will not stop the simulation regardless of the severity level of the assertion.

WLF Files tab



The **WLF Files** tab includes these options:

- **WLF File Size Limit**
  Limits the WLF file by size (as closely as possible) to the specified number of megabytes. If both size and time limits are specified, the most restrictive is used. Setting it to 0 results in no limit. Edit the WLFSizeLimit (UM-348) variable in the *modelsim.ini* file to set a permanent default.
- **WLF File Time Limit**
  Limits the WLF file by size (as closely as possible) to the specified amount of time. If both time and size limits are specified, the most restrictive is used. Setting it to 0 results in no limit. Edit the WLFTimeLimit (UM-348) variable in the *modelsim.ini* file to set a permanent default.
- **Compress WLF data**
  Compresses WLF files to reduce their size. You would typically only disable compression for troubleshooting purposes. Edit the WLFCompress (UM-348) variable in the *modelsim.ini* file to set a permanent default.

- **Delete WLF file on exit**
  Specifies whether the WLF file should be deleted when the simulation ends. Edit the WLFDeleteOnQuit (UM-348) variable in the *modelsim.ini* file to set a permanent default.
- **Design Hierarchy**
  Specifies whether to save all design hierarchy in the WLF file or only regions containing logged signals. Edit the WLFSaveAllRegions (UM-348) variable in the *modelsim.ini* file to set a permanent default.

Creating and managing breakpoints

ModelSim supports both signal (i.e., when conditions) and file-line breakpoints. Breakpoints can be set from multiple locations in the GUI or from the command line.

Signal breakpoints

Signal breakpoints (when conditions) instruct ModelSim to perform actions when the specified conditions are met. For example, you can break on a signal value or at a specific simulator time (see the **when** command (CR-205) for additional details). When a breakpoint is hit, a message in the Main window transcript identifies the signal that caused the breakpoint.

Setting signal breakpoints from the command line

You use the **when** command (CR-205) to set a signal breakpoint from the VSIM> prompt. See the *Command Reference* for further details.

Setting signal breakpoints from the GUI

Signal breakpoints are most easily set in the Signals window (UM-183) and the Wave window (UM-206). Select a signal, click your right mouse button, and select **Insert Breakpoint** from the context menu. A breakpoint is set on that signal and will be listed in the **Breakpoints** dialog.

Alternatively you can set signal breakpoints from the Breakpoints dialog (UM-259).

File-line breakpoints

File-line breakpoints are set on executable lines in your source files. When the line is hit, the simulator stops.

Setting file-line breakpoints from the command line

You use the **bp** command (CR-46) to set a file-line breakpoint from the VSIM> prompt. See the *Command Reference* for further details.

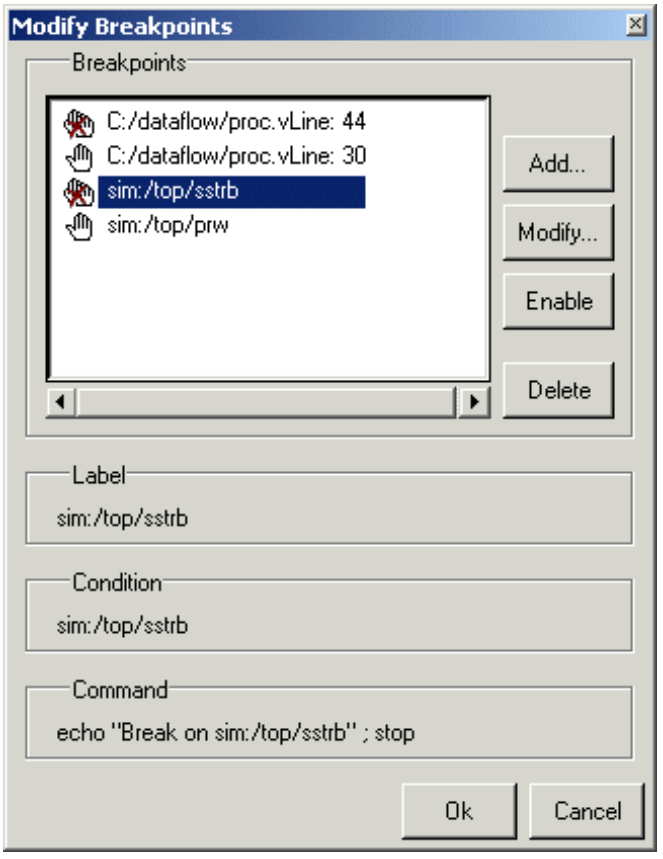
Setting file-line breakpoints from the GUI

File-line breakpoints are most easily set using your mouse in the Source window (UM-191). Click on a blue line number at the left side of the Source window, and a red diamond denoting a breakpoint will appear. The breakpoints are toggles – click once to create the colored diamond; click again to disable or enable the breakpoint. To delete the breakpoint completely, click the red diamond with your right mouse button, and select **Remove Breakpoint**.

Alternatively you can set file-line breakpoints from the Breakpoints dialog (UM-259).

Breakpoints dialog

The Breakpoints dialog box allows you to create and manage both Signal breakpoints (UM-258) and File-line breakpoints (UM-258). Select **Tools > Breakpoints** from the Main, Signals, Source, or Wave windows to open the dialog.



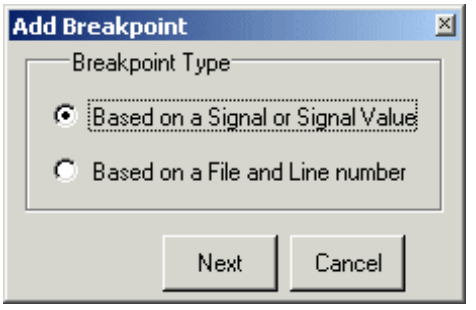
The **Breakpoints** dialog includes these options:

- **Breakpoints**
  List of all existing breakpoints. Breakpoints set from anywhere in the GUI, or from the command line, are listed. A red 'X' through the hand icon means the breakpoint is currently disabled.
- **Add**
  Create a new signal or file-line breakpoint. See below for more details.
- **Modify**
  Change properties of an existing breakpoint. See below for more details.
- **Disable/Enable**
  De-activate or activate the selected breakpoint.
- **Delete**
  Delete the selected breakpoint.
- **Label**
  Text label of the selected breakpoint.

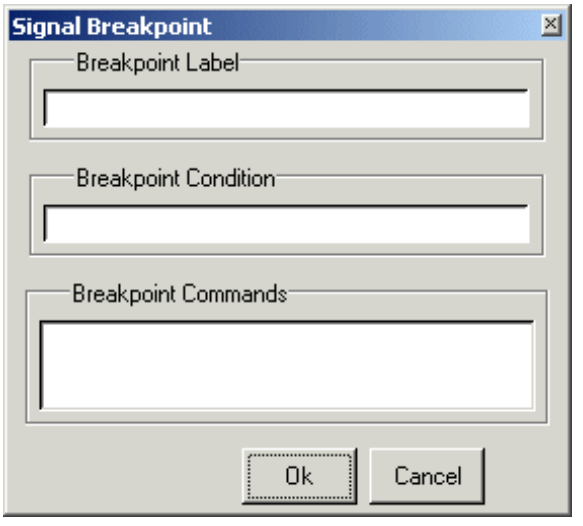
- **Condition**
  The condition under which the breakpoint will be hit.
- **Command**
  The command that will be executed when the breakpoint is hit.

Adding a breakpoint

Click Add to add a new breakpoint, and you will see the Add Breakpoint dialog.



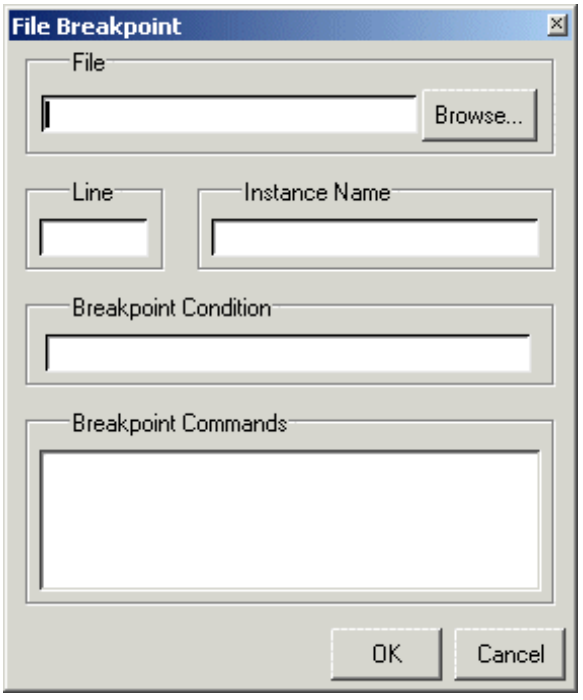
Choose whether to create a signal breakpoint or a file-line breakpoint and then select Next. Depending on which type of breakpoint you're creating, you'll see one of the two dialogs below. These are the same dialogs you'll see if you modify an exiting breakpoint.



The **Signals Breakpoint** dialog includes these options:

- **Breakpoint Label**
  Specify an optional text label for the breakpoint.
- **Breakpoint Condition**
  Specify condition(s) to be met for the command(s) to be executed. See the **when** command (CR-205) for more information on creating the condition statement.

- **Breakpoint Commands**
  Specify command(s) to be executed when the condition is met. Any ModelSim or Tcl command or series of commands are valid, with one exception – the **run** command (CR-114) cannot be used.



The **File Breakpoint** dialog includes these options:

- **File**
  Specify the file in which to set the breakpoint.
- **Line**
  Specify the line number on which to set the breakpoint. Note that breakpoints can be set only on executable lines.
- **Instance Name**
  Specify a region in which to apply the breakpoint. If left blank the breakpoint affects every instance in the design.
- **Breakpoint Condition**
  Specify a condition that determines whether the breakpoint is hit.
- **Breakpoint Commands**
  Specify command(s) to be executed when the breakpoint is hit. Any ModelSim or Tcl command or series of commands is valid, with one exception – the **run** command (CR-114) cannot be used.

Miscellaneous tools and add-ons

Several miscellaneous tools and add-ons are available from ModelSim menus. Follow the links below for more information.

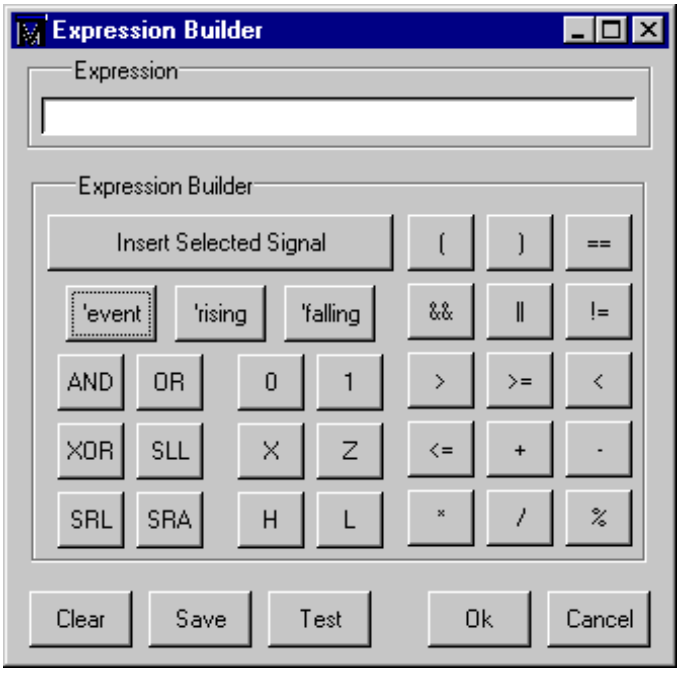
- The GUI Expression Builder (UM-262)
  Edit > Search > Search for Expression > Builder (List or Wave window)
  Helps you build logical expressions for use in Wave and List window searches and several simulator commands. For expression format syntax see "GUI_expression_format" (CR-15).
- Language templates (UM-264)
  View > Show language templates (Source window)
  Helps you write VHDL or Verilog code

The GUI Expression Builder

The GUI Expression Builder is a feature of the Wave and List Signal Search dialog boxes, and the List trigger properties dialog box. It aids in building a search expression that follows the "GUI_expression_format" (CR-15).

To locate the Builder:

- select **Edit > Search** (List or Wave window)
- select the **Search for Expression** option in the resulting dialog box
- select the **Builder** button



The Expression Builder dialog box provides an array of buttons that help you build a GUI expression. For instance, rather than typing in a signal name, you can select the signal in the associated Wave or List window and press Insert Reference Signal in the Expression

Builder. The result will be the full signal name added to the expression field. All Expression Builder buttons correspond to the "Expression syntax" (CR-18).

To search for when a signal reaches a particular value

Select the signal in the Wave window and click **Insert Selected Signal** and ==. Then, click the value buttons or type a value.

To evaluate only on clock edges

Click the **&&** button to AND this condition with the rest of the expression. Then select the clock in the Wave window and click **Insert Selected Signal** and **'rising**. You can also select the falling edge or both edges.

Operators

Other buttons will add operators of various kinds (see "Expression syntax" (CR-18)), or you can type them in.

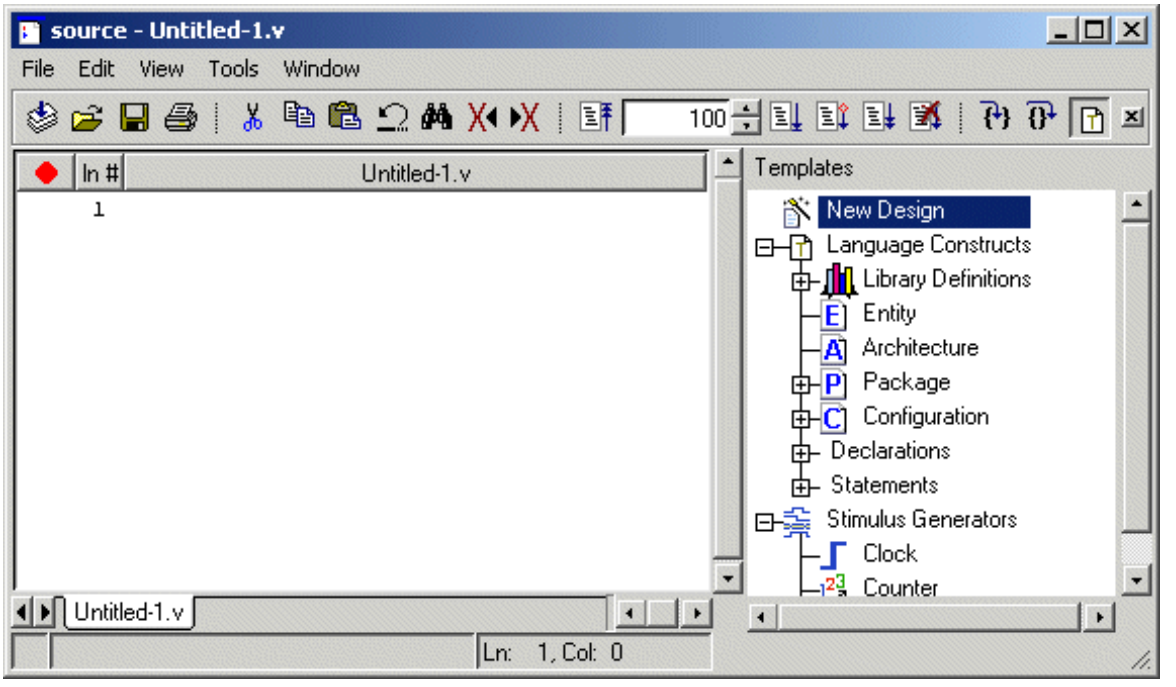
See "Configuring a List trigger with Expression Builder" (UM-382) for an additional Expression builder example.

Language templates

ModelSim language templates help you write VHDL or Verilog code. They are a collection of wizards, menus, and dialogs that produce code for new designs, language constructs, logic blocks, etc.

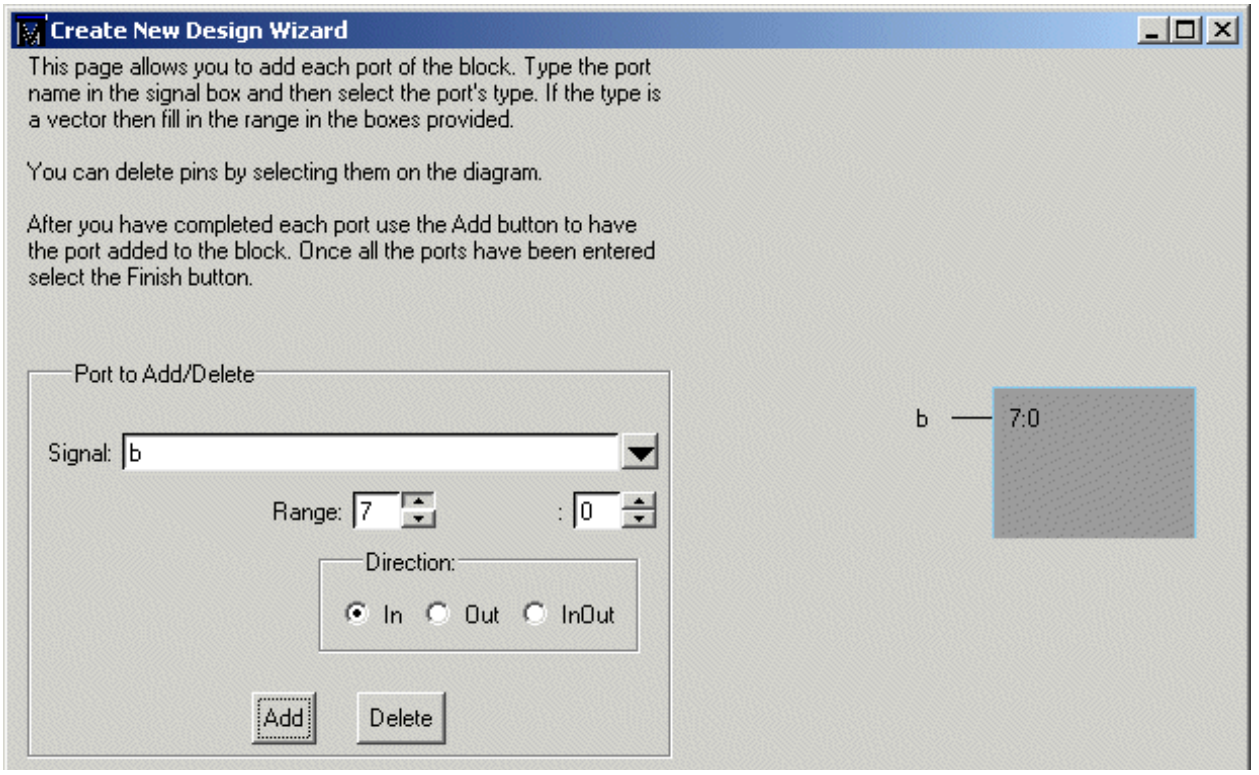
Important: The language templates are not intended to replace thorough knowledge of HDL coding. They are intended as an interactive "reference" for creating small sections of code. If you are unfamiliar with VHDL or Verilog, you should attend a training class or consult one of the many books available on HDL languages.

To use the templates, either open an existing HDL file in the Source window (UM-191), or select **File > New** (Source window) to create a new file. Once the file is open, select **View > Show language templates** . This displays a pane that shows the available templates.

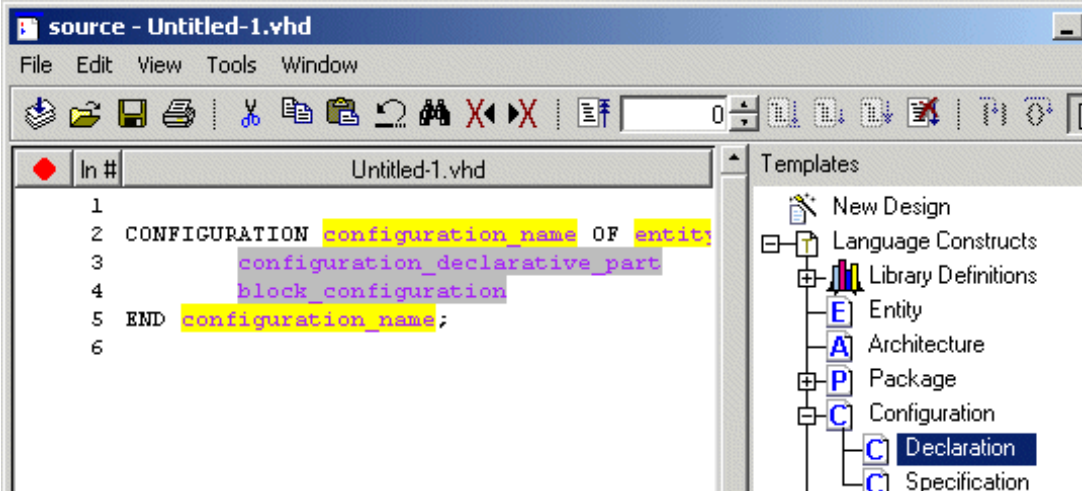


The templates that appear depend on the type of file you create. For example Module and Primitive templates are available for Verilog files, and Entity and Architecture templates are available for VHDL files.

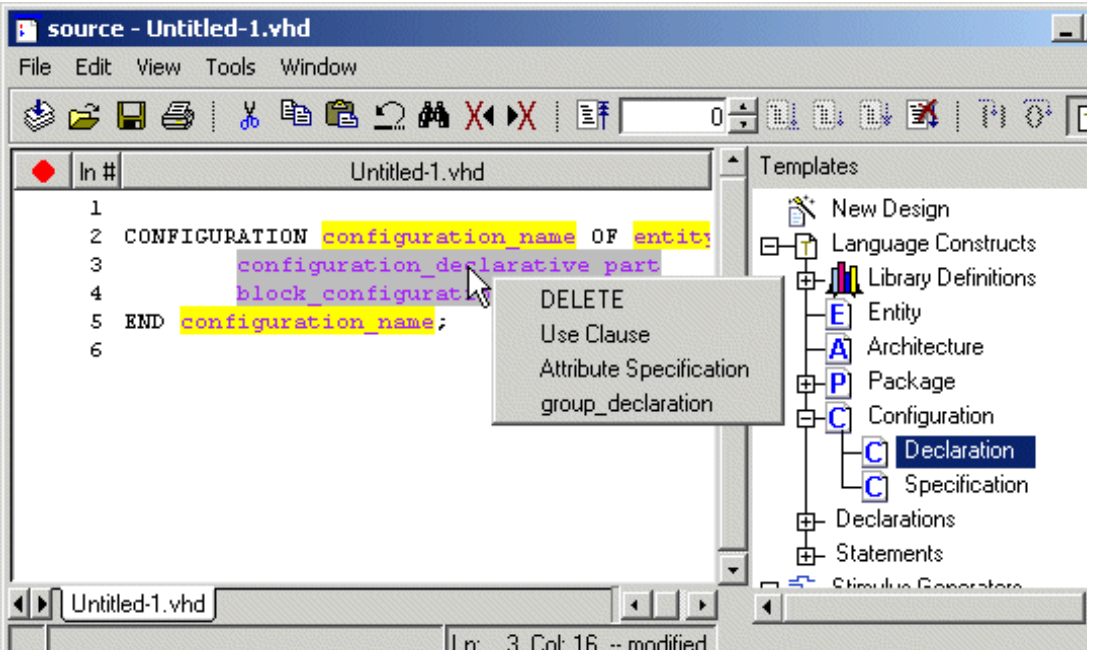
Double-click an item in the list to begin creating code. Some of the items bring up wizards while others insert code into your HDL file. The dialog below is part of the wizard for creating a new design. Simply follow the directions in the wizards.


Create New Design Wizard
This page allows you to add each port of the block. Type the port name in the signal box and then select the port's type. If the type is a vector then fill in the range in the boxes provided.
You can delete pins by selecting them on the diagram.
After you have completed each port use the Add button to have the port added to the block. Once all the ports have been entered select the Finish button.
Port to Add/Delete
Signal: b
Range: 7 : 0
Direction:
In Out InOut
Add
Delete
b
7:0

Code inserted into your source file may contain yellow or gray highlighted "fields". Yellow highlighting identifies an object that needs a name. Double-click the yellow object to enter a name. Note that all yellow objects with the same label (e.g., "configuration_name" below) will change to whatever name you enter. This ensures matching fields remain in synch.



Gray highlighting indicates that a context menu with additional commands is available. In the example below, right-clicking "configuration_declarative_part" gives you three options for continuing the definition of the Configuration.



The first menu item is always "DELETE." This allows you to remove unwanted objects from the HDL code, such as optional fields.

Keyboard shortcut

<control - p> edits a yellow field and expands a gray field.

Graphic interface commands

The following commands provide control and feedback during simulation. Only brief descriptions are provided here; for more information and command syntax see the *ModelSim Command Reference*.

Window control and feedback commands	Description
batch_mode (CR-40)	returns a 1 if ModelSim is operating in batch mode, otherwise returns a 0; it is typically used as a condition in an if statement
configure (CR-51)	invokes the List or Wave widget configure command for the current default List or Wave window
notepad (CR-95)	a simple text editor; used to view and edit ASCII files or create new files
write preferences (CR-219)	saves the current GUI preference settings to a Tcl preference file

8 - Signal Spy

Chapter contents

Introduction UM-270
Designed for testbenches UM-270
init_signal_driver UM-271
init_signal_spy UM-274
signal_force UM-276
signal_release UM-278
$init_signal_driver UM-280
$init_signal_spy UM-283
$signal_force UM-285
$signal_release UM-287

This chapter describes the Signal Spy™ procedures and system tasks. These allow you to monitor, drive, force, and release hierarchical items in VHDL or mixed designs.

Introduction

The Verilog language allows access to any signal from any other hierarchical block without having to route it via the interface. This means you can use hierarchical notation to either assign or determine the value of a signal in the design hierarchy from a testbench. This capability fails when a Verilog testbench attempts to reference a signal in a VHDL block or reference a signal in a Verilog block through a VHDL level of hierarchy.

This limitation exists because VHDL does not allow hierarchical notation. In order to reference internal hierarchical signals, you have to resort to defining signals in a global package and then utilize those signals in the hierarchical blocks in question. But, this requires that you keep making changes depending on the signals that you want to reference.

The Signal Spy procedures and system tasks overcome the aforementioned limitations. They allow you to monitor (spy), drive, force, or release hierarchical objects in a VHDL or mixed design.

The VHDL procedures are provided via the "Util package" (UM-62) within the *modelsim_lib* library. To access the procedures you would add lines like the following to your VHDL code:

```
library modelsim_lib;
use modelsim_lib.util.all;
```

The Verilog tasks are available as built-in "System tasks" (UM-89). The table below shows the VHDL procedures and their corresponding Verilog system tasks.

VHDL procedures	Verilog system tasks
init_signal_driver (UM-271)	$init_signal_driver (UM-280)
init_signal_spy (UM-274)	$init_signal_spy (UM-283)
signal_force (UM-276)	$signal_force (UM-285)
signal_release (UM-278)	$signal_release (UM-287)

Designed for testbenches

Signal Spy limits the portability of your code. HDL code with Signal Spy procedures or tasks works only in ModelSim, not other simulators. We therefore recommend using Signal Spy only in testbenches, where portability is less of a concern, and the need for such a tool is more applicable.

init_signal_driver

The init_signal_driver() procedure drives the value of a VHDL signal or Verilog net (called the src_object) onto an existing VHDL signal or Verilog net (called the dest_object). This allows you to drive signals or nets at any level of the design hierarchy from within a VHDL architecture (e.g., a testbench).

The init_signal_driver procedure drives the value onto the destination signal just as if the signals were directly connected in the HDL code. Any existing or subsequent drive or force of the destination signal, by some other means, will be considered with the init_signal_driver value in the resolution of the signal.

Call only once

The init_signal_driver procedure creates a persistent relationship between the source and destination signals. Hence, you need to call init_signal_driver only once for a particular pair of signals. Once init_signal_driver is called, any change on the source signal will be driven on the destination signal until the end of the simulation.

Thus, we recommend that you place all init_signal_driver calls in a VHDL process. You need to code the VHDL process correctly so that it is executed only once. The VHDL process should not be sensitive to any signals and should contain only init_signal_driver calls and a simple wait statement. The process will execute once and then wait forever. See the example below.

Syntax

```
init_signal_driver(src_object, dest_object, delay, delay_type, verbose)
```

Returns

Nothing

Arguments

Name	Type	Description
src_object	string	Required. A full hierarchical path (or relative path with reference to the calling block) to a VHDL signal or Verilog net. Use the path separator to which your simulation is set (i.e., "/" or "."). A full hierarchical path must begin with a "/" or ".". The path must be contained within double quotes.
dest_object	string	Required. A full hierarchical path (or relative path with reference to the calling block) to an existing VHDL signal or Verilog net. Use the path separator to which your simulation is set (i.e., "/" or "."). A full hierarchical path must begin with a "/" or ".". The path must be contained within double quotes.
delay	time	Optional. Specifies a delay relative to the time at which the src_object changes. The delay can be an inertial or transport delay. If no delay is specified, then a delay of zero is assumed.
delay_type	del_mode	Optional. Specifies the type of delay that will be applied. The value must be either mti_inertial or mti_transport. The default is mti_inertial.
verbose	integer	Optional. Possible values are 0 or 1. Specifies whether you want a message reported in the Transcript stating that the src_object is driving the dest_object. Default is 0, no message.

Related procedures

init_signal_spy (UM-274), signal_force (UM-276), signal_release (UM-278)

Limitations

- When driving a Verilog net, the only *delay_type* allowed is inertial. If you set the delay type to *mti_transport*, the setting will be ignored and the delay type will be *mti_inertial*.
- Any delays that are set to a value less than the simulator resolution will be rounded to the nearest resolution unit; no special warning will be issued.

Example

```
library IEEE, modelsim_lib;
use IEEE.std_logic_1164.all;
use modelsim_lib.util.all;

entity testbench is
end;

architecture only of testbench is
  signal clk0 : std_logic;

begin

  gen_clk0 : process
  begin
    clk0 <= '1' after 0 ps, '0' after 20 ps;
    wait for 40 ps;
  end process gen_clk0;

  drive_sig_process : process
  begin
    init_signal_driver("clk0", "/testbench/uut/blk1/clk", open, open, 1);
    init_signal_driver("clk0", "/testbench/uut/blk2/clk", 100 ps, \
    mti_transport);
    wait;
  end process drive_sig_process;

  ...

end;
```

The above example creates a local clock (*clk0*) and connects it to two clocks within the design hierarchy. The .../*blk1/clk* will match local *clk0* and a message will be displayed. The *open* entries allow the default delay and delay_type while setting the verbose parameter to a 1. The .../*blk2/clk* will match the local *clk0* but be delayed by 100 ps.

init_signal_spy

The init_signal_spy() procedure mirrors the value of a VHDL signal or Verilog register/net (called the src_object) onto an existing VHDL signal or Verilog register/net (called the dest_object). This allows you to reference signals, registers, or nets at any level of hierarchy from within a VHDL architecture (e.g., a testbench).

The init_signal_spy procedure only sets the value onto the destination signal and does not drive or force the value. Any existing or subsequent drive or force of the destination signal, by some other means, will override the value that was set by init_signal_spy.

Call only once

The init_signal_spy procedure creates a persistent relationship between the source and destination signals. Hence, you need to call init_signal_spy once for a particular pair of signals. Once init_signal_spy is called, any change on the source signal will mirror on the destination signal until the end of the simulation.

Thus, we recommend that you place all init_signal_spy calls in a VHDL process. You need to code the VHDL process correctly so that it is executed only once. The VHDL process should not be sensitive to any signals and should contain only init_signal_spy calls and a simple wait statement. The process will execute once and then wait forever, which is the desired behavior. See the example below.

Syntax

```
init_signal_spy(src_object, dest_object, verbose)
```

Returns

Nothing

Arguments

Name	Type	Description
src_object	string	Required. A full hierarchical path (or relative path with reference to the calling block) to a VHDL signal or Verilog register/net. Use the path separator to which your simulation is set (i.e., "/" or "."). A full hierarchical path must begin with a "/" or ".". The path must be contained within double quotes.

Name	Type	Description
dest_object	string	Required. A full hierarchical path (or relative path with reference to the calling block) to an existing VHDL signal or Verilog register. Use the path separator to which your simulation is set (i.e., "/" or "."). A full hierarchical path must begin with a "/" or ".". The path must be contained within double quotes.
verbose	integer	Optional. Possible values are 0 or 1. Specifies whether you want a message reported in the Transcript stating that the spy_object's value is mirrored onto the dest_object. Default is 0, no message.

Related functions

init_signal_driver (UM-271), signal_force (UM-276), signal_release (UM-278)

Limitations

- When mirroring the value of a Verilog register/net onto a VHDL signal, the VHDL signal must be of type bit, bit_vector, std_logic, or std_logic_vector.
- Verilog memories (arrays of registers) are not supported.

Example

```
library ieee, modelsim_lib;
use ieee.std_logic_1164.all
use modelsim_lib.util.all;
entity top is
end;

architecture only of top is
  signal top_sig1 : std_logic;
begin
  ...
  spy_process : process
  begin
    init_signal_spy("/top/uut/inst1/sig1","/top_sig1",1);
    wait;
  end process spy_process;
  ...
end;
```

In this example, the value of */top/uut/inst1/sig1* will be mirrored onto */top_sig1*.

signal_force

The signal_force() procedure forces the value specified onto an existing VHDL signal or Verilog register or net (called the dest_object). This allows you to force signals, registers, or nets at any level of the design hierarchy from within a VHDL architecture (e.g., a testbench).

A signal_force works the same as the **force** command (CR-82) with the exception that you cannot issue a repeating force. The force will remain on the signal until a signal_release, a force or release command, or a subsequent signal_force is issued. Signal_force can be called concurrently or sequentially in a process.

Syntax

```
signal_force( dest_object, value, rel_time, force_type, cancel_period,
verbose )
```

Returns

Nothing

Arguments

Name	Type	Description
dest_object	string	Required. A full hierarchical path (or relative path with reference to the calling block) to an existing VHDL signal or Verilog register/net. Use the path separator to which your simulation is set (i.e., "/" or "."). A full hierarchical path must begin with a "/" or ".". The path must be contained within double quotes.
value	string	Required. Specifies the value to which the dest_object is to be forced. The specified value must be appropriate for the type.
rel_time	time	Optional. Specifies a time relative to the current simulation time for the force to occur. The default is 0.
force_type	forcetype	Optional. Specifies the type of force that will be applied. The value must be one of the following; default, deposit, drive, or freeze. The default is "default" (which is "freeze" for unresolved objects or "drive" for resolved objects). See the **force** command (CR-82) for further details on force type.

Name	Type	Description
cancel_period	time	Optional. Cancels the signal_force command after the specified period of time units. Cancellation occurs at the last simulation delta cycle of a time unit. A value of zero cancels the force at the end of the current time period. Default is -1 ms. A negative value means that the force will not be cancelled.
verbose	integer	Optional. Possible values are 0 or 1. Specifies whether you want a message reported in the Transcript stating that the value is being forced on the dest_object at the specified time. Default is 0, no message.

Related functions

init_signal_driver (UM-271), init_signal_spy (UM-274), signal_release (UM-278)

Limitations

You cannot force bits or slices of a register; you can force only the entire register.

Example

```
library IEEE, modelsim_lib;
use IEEE.std_logic_1164.all;
use modelsim_lib.util.all;

entity testbench is
end;

architecture only of testbench is
begin

  force_process : process
  begin
    signal_force("/testbench/uut/blk1/reset", "1", 0 ns, freeze, open, 1);
    signal_force("/testbench/uut/blk1/reset", "0", 40 ns, freeze, 2 ms, 1);
    wait;
  end process force_process;

  ...

end;
```

The above example forces *reset* to a "1" from time 0 ns to 40 ns. At 40 ns, *reset* is forced to a "0", 2 ms after the second signal_force call was executed.

If you want to skip parameters so that you can specify subsequent parameters, you need to use the keyword "open" as a placeholder for the skipped parameter(s). The first signal_force procedure illustrates this, where an "open" for the cancel_period parameter means that the default value of -1 ms is used.

signal_release

The signal_release() procedure releases any force that was applied to an existing VHDL signal or Verilog register/net (called the dest_object). This allows you to release signals, registers or nets at any level of the design hierarchy from within a VHDL architecture (e.g., a testbench).

A signal_release works the same as the **noforce** command (CR-92). Signal_release can be called concurrently or sequentially in a process.

Syntax

```
signal_release( dest_object, verbose )
```

Returns

Nothing

Arguments

Name	Type	Description
dest_object	string	Required. A full hierarchical path (or relative path with reference to the calling block) to an existing VHDL signal or Verilog register/net. Use the path separator to which your simulation is set (i.e., "/" or "."). A full hierarchical path must begin with a "/" or ".". The path must be contained within double quotes.
verbose	integer	Optional. Possible values are 0 or 1. Specifies whether you want a message reported in the Transcript stating that the signal is being released and the time of the release. Default is 0, no message.

Related functions

init_signal_driver (UM-271), init_signal_spy (UM-274), signal_force (UM-276)

Limitations

- You cannot release a bit or slice of a register; you can release only the entire register.

Example

```
library IEEE, modelsim_lib;
use IEEE.std_logic_1164.all;
use modelsim_lib.util.all;

entity testbench is
end;

architecture only of testbench is

  signal release_flag : std_logic;

begin

  stim_design : process
  begin
    ...
    wait until release_flag = '1';
    signal_release("/testbench/dut/blk1/data", 1);
    signal_release("/testbench/dut/blk1/clk", 1);
    ...
  end process stim_design;

  ...

end;
```

The above example releases any forces on the signals data and *clk* when the signal *release_flag* is a "1". Both calls will send a message to the transcript stating which signal was released and when.

$init_signal_driver

The $init_signal_driver() system task drives the value of a VHDL signal or Verilog register/net (called the src_object) onto an existing VHDL signal or Verilog net (called the dest_object). This allows you to drive signals or nets at any level of the design hierarchy from within a Verilog module (e.g., a testbench).

The $init_signal_driver system task drives the value onto the destination signal just as if the signals were directly connected in the HDL code. Any existing or subsequent drive or force of the destination signal, by some other means, will be considered with the $init_signal_driver value in the resolution of the signal.

Call only once

The $init_signal_driver system task creates a persistent relationship between the source and destination signals. Hence, you need to call $init_signal_driver only once for a particular pair of signals. Once $init_signal_driver is called, any change on the source signal will be driven on the destination signal until the end of the simulation.

Thus, we recommend that you place all $init_signal_driver calls in a Verilog initial block. See the example below.

Syntax

```
$init_signal_driver(src_object, dest_object, delay, delay_type, verbose)
```

Returns

Nothing

Arguments

Name	Type	Description
src_object	string	Required. A full hierarchical path (or relative path with reference to the calling block) to a VHDL signal or Verilog net. Use the path separator to which your simulation is set (i.e., "/" or "."). A full hierarchical path must begin with a "/" or ".". The path must be contained within double quotes.
dest_object	string	Required. A full hierarchical path (or relative path with reference to the calling block) to an existing VHDL signal or Verilog net. Use the path separator to which your simulation is set (i.e., "/" or "."). A full hierarchical path must begin with a "/" or ".". The path must be contained within double quotes.

Name	Type	Description
delay	integer, real, or time	Optional. Specifies a delay relative to the time at which the src_object changes. The delay can be an inertial or transport delay. If no delay is specified, then a delay of zero is assumed.
delay_type	integer	Optional. Specifies the type of delay that will be applied. The value must be either 0 (inertial) or 1 (transport). The default is 0.
verbose	integer	Optional. Possible values are 0 or 1. Specifies whether you want a message reported in the Transcript stating that the src_object is driving the dest_object. Default is 0, no message.

Related procedures

$init_signal_spy (UM-283), $signal_force (UM-285), $signal_release (UM-287)

Limitations

- When driving a Verilog net, the only *delay_type* allowed is inertial. If you set the delay type to 1 (transport), the setting will be ignored, and the delay type will be inertial.
- Any delays that are set to a value less than the simulator resolution will be rounded to the nearest resolution unit; no special warning will be issued.

Example

```
‘timescale 1 ps / 1 ps

module testbench;

reg clk0;

initial begin
  clk0 = 1;
  forever begin
   #20 clk0 = ~clk0;
  end
end

initial begin
   $init_signal_driver("clk0", "/testbench/uut/blk1/clk", , , 1);
   $init_signal_driver("clk0", "/testbench/uut/blk2/clk", 100, 1);
end

  ...

endmodule
```

The above example creates a local clock (*clk0*) and connects it to two clocks within the design hierarchy. The *.../blk1/clk* will match local *clk0* and a message will be displayed. The *.../blk2/clk* will match the local *clk0* but be delayed by 100 ps. For the second call to work, the *.../blk2/clk* must be a VHDL based signal, because if it were a Verilog net a 100 ps inertial delay would consume the 40 ps clock period. Verilog nets are limited to only inertial delays and thus the setting of 1 (transport delay) would be ignored.

$init_signal_spy

The $init_signal_spy() system task mirrors the value of a VHDL signal or Verilog register/net (called the src_object) onto an existing VHDL signal or Verilog register/net (called the dest_object). This allows you to reference signals, registers, or nets at any level of hierarchy from within a Verilog module (e.g., a testbench).

The $init_signal_spy system task only sets the value onto the destination signal and does not drive or force the value. Any existing or subsequent drive or force of the destination signal, by some other means, will override the value set by $init_signal_spy.

Call only once

The $init_signal_spy system task creates a persistent relationship between the source and the destination signal. Hence, you need to call $init_signal_spy only once for a particular pair of signals. Once $init_signal_spy is called, any change on the source signal will mirror on the destination signal until the end of the simulation. Thus, we recommend that you place all $init_signal_spy calls in a Verilog initial block. See the example below.

Syntax

```
$init_signal_spy(src_object, dest_object, verbose)
```

Returns

Nothing

Arguments

Name	Type	Description
src_object	string	Required. A full hierarchical path (or relative path with reference to the calling block) to a VHDL signal or Verilog register/net. Use the path separator to which your simulation is set (i.e., "/" or "."). A full hierarchical path must begin with a "/" or ".". The path must be contained within double quotes.

Name	Type	Description
dest_object	string	Required. A full hierarchical path (or relative path with reference to the calling block) to a Verilog register or VHDL signal. Use the path separator to which your simulation is set (i.e., "/" or "."). A full hierarchical path must begin with a "/" or ".". The path must be contained within double quotes.
verbose	integer	Optional. Possible values are 0 or 1. Specifies whether you want a message reported in the Transcript stating that the spy_object's value is mirrored onto the dest_object. Default is 0, no message.

Related tasks

$init_signal_driver (UM-280), $signal_force (UM-285), $signal_release (UM-287)

Limitations

- When mirroring the value of a VHDL signal onto a Verilog register, the VHDL signal must be of type bit, bit_vector, std_logic, or std_logic_vector.
- Verilog memories (arrays of registers) are not supported.

Example

```
module testbench;
...
reg top_sig1;
...
initial
  begin
    $init_signal_spy("/top/uut/inst1/sig1","/top_sig1", 1);
  end
...
endmodule
```

In this example, the value of */top/uut/inst1/sig1* will be mirrored onto */top_sig1*.

$signal_force

The $signal_force() system task forces the value specified onto an existing VHDL signal or Verilog register/net (called the dest_object). This allows you to force signals, registers, or nets at any level of the design hierarchy from within a Verilog module (e.g., a testbench).

A $signal_force works the same as the **force** command (CR-82) with the exception that you cannot issue a repeating force. The force will remain on the signal until a $signal_release, a force or release command, or a subsequent $signal_force is issued. $signal_force can be called concurrently or sequentially in a process.

Syntax

```
$signal_force( dest_object, value, rel_time, force_type, cancel_period,
verbose )
```

Returns

Nothing

Arguments

Name	Type	Description
dest_object	string	Required. A full hierarchical path (or relative path with reference to the calling block) to an existing VHDL signal or Verilog register/net. Use the path separator to which your simulation is set (i.e., "/" or "."). A full hierarchical path must begin with a "/" or ".". The path must be contained within double quotes.
value	string	Required. Specifies the value to which the dest_object is to be forced. The specified value must be appropriate for the type.
rel_time	integer, real, or time	Optional. Specifies a time relative to the current simulation time for the force to occur. The default is 0.
force_type	integer	Optional. Specifies the type of force that will be applied. The value must be one of the following; 0 (default), 1 (deposit), 2 (drive), or 3 (freeze). The default is "default" (which is "freeze" for unresolved objects or "drive" for resolved objects). See the **force** command (CR-82) for further details on force type.

Name	Type	Description
cancel_period	integer, real, time	Optional. Cancels the $signal_force command after the specified period of time units. Cancellation occurs at the last simulation delta cycle of a time unit. A value of zero cancels the force at the end of the current time period. Default is -1. A negative value means that the force will not be cancelled.
verbose	integer	Optional. Possible values are 0 or 1. Specifies whether you want a message reported in the Transcript stating that the value is being forced on the dest_object at the specified time. Default is 0, no message.

Related functions

$init_signal_driver (UM-280), $init_signal_spy (UM-283), $signal_release (UM-287)

Limitations

You cannot force bits or slices of a register; you can force only the entire register.

Example

```
‘timescale 1 ns / 1 ns

module testbench;

initial
  begin
   $signal_force("/testbench/uut/blk1/reset", "1", 0, 3, , 1);
   $signal_force("/testbench/uut/blk1/reset", "0", 40, 3, 200000, 1);
  end

...

endmodule
```

The above example forces *reset* to a "1" from time 0 ns to 40 ns. At 40 ns, *reset* is forced to a "0", 200000 ns after the second $signal_force call was executed.

$signal_release

The $signal_release() system task releases any force that was applied to an existing VHDL signal or Verilog register/net (called the dest_object). This allows you to release signals, registers, or nets at any level of the design hierarchy from within a Verilog module (e.g., a testbench).

A $signal_release works the same as the **noforce** command (CR-92). $signal_release can be called concurrently or sequentially in a process.

Syntax

```
$signal_release( dest_object, verbose )
```

Returns

Nothing

Arguments

Name	Type	Description
dest_object	string	Required. A full hierarchical path (or relative path with reference to the calling block) to an existing VHDL signal or Verilog register/net. Use the path separator to which your simulation is set (i.e., "/" or "."). A full hierarchical path must begin with a "/" or ".". The path must be contained within double quotes.
verbose	integer	Optional. Possible values are 0 or 1. Specifies whether you want a message reported in the Transcript stating that the signal is being released and the time of the release. Default is 0, no message.

Related functions

$init_signal_driver (UM-280), $init_signal_spy (UM-283), $signal_force (UM-285)

Limitations

- You cannot release a bit or slice of a register; you can release only the entire register.

Example

```
module testbench;

reg release_flag;

always @(posedge release_flag) begin
  $signal_release("/testbench/dut/blk1/data", 1);
  $signal_release("/testbench/dut/blk1/clk", 1);
end

...

endmodule
```

The above example releases any forces on the signals *data* and *clk* when the register *release_flag* transitions to a "1". Both calls will send a message to the transcript stating which signal was released and when.

9 - Standard Delay Format (SDF) Timing Annotation

Chapter contents

Specifying SDF files for simulation UM-290
Instance specification UM-290
SDF specification with the GUI UM-291
Errors and warnings UM-291

VHDL VITAL SDF UM-292
SDF to VHDL generic matching UM-292
Resolving errors UM-293

Verilog SDF UM-294
The $sdf_annotate system task UM-294
SDF to Verilog construct matching UM-295
Optional edge specifications UM-298
Optional conditions UM-299
Rounded timing values UM-299

SDF for Mixed VHDL and Verilog Designs UM-300

Interconnect delays. UM-300

Disabling timing checks UM-300

Troubleshooting UM-301
Specifying the wrong instance UM-301
Mistaking a component or module name for an instance label . UM-302
Forgetting to specify the instance UM-302

This chapter discusses ModelSim's implementation of SDF (Standard Delay Format) timing annotation. Included are sections on VITAL SDF and Verilog SDF, plus troubleshooting.

Verilog and VHDL VITAL timing data can be annotated from SDF files by using the simulator's built-in SDF annotator.

SDF and ModelSim

SDF timing annotations can be applied only to your FPGA vendor's libraries; all other libraries will simulate without annotation.

Specifying SDF files for simulation

ModelSim supports SDF versions 1.0 through 3.0. The simulator's built-in SDF annotator automatically adjusts to the version of the file. Use the following **vsim** (CR-189) command-line options to specify the SDF files, the desired timing values, and their associated design instances:

```
-sdfmin [<instance>=]<filename>
-sdftyp [<instance>=]<filename>
-sdfmax [<instance>=]<filename>
```

Any number of SDF files can be applied to any instance in the design by specifying one of the above options for each file. Use **-sdfmin** to select minimum, **-sdftyp** to select typical, and **-sdfmax** to select maximum timing values from the SDF file.

Instance specification

The instance paths in the SDF file are relative to the instance to which the SDF is applied. Usually, this instance is an ASIC or FPGA model instantiated under a testbench. For example, to annotate maximum timing values from the SDF file *myasic.sdf* to an instance *u1* under a top-level named *testbench*, invoke the simulator as follows:

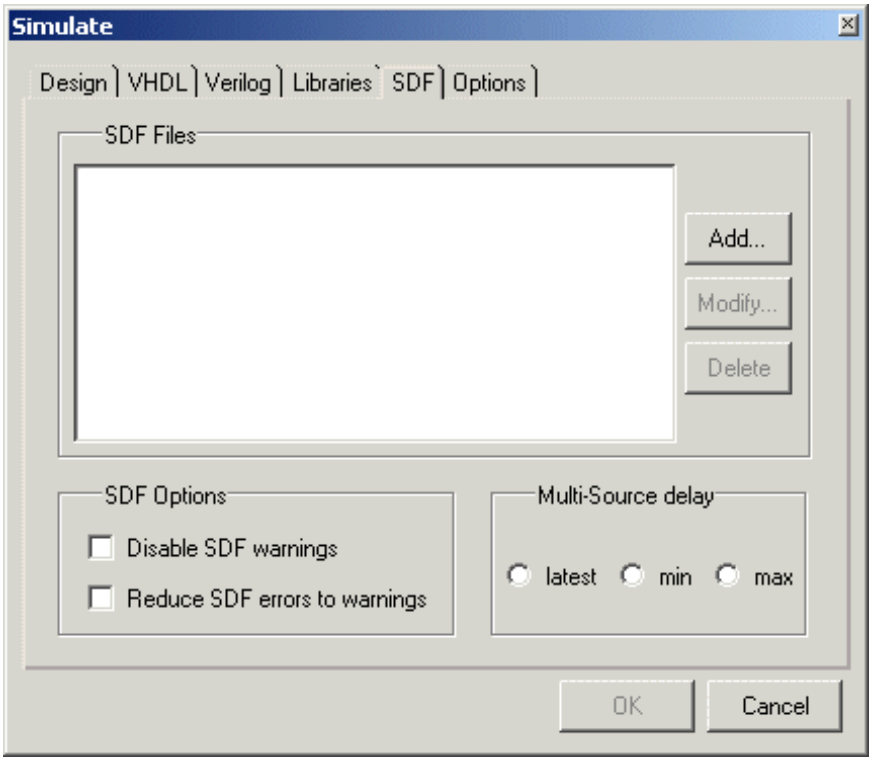
```
vsim -sdfmax /testbench/u1=myasic.sdf testbench
```

If the instance name is omitted then the SDF file is applied to the top-level. *This is usually incorrect* because in most cases the model is instantiated under a testbench or within a larger system level simulation. In fact, the design can have several models, each having its own SDF file. In this case, specify an SDF file for each instance. For example,

```
vsim -sdfmax /system/u1=asic1.sdf -sdfmax /system/u2=asic2.sdf system
```

SDF specification with the GUI

As an alternative to the command-line options, you can specify SDF files in the **Simulate** dialog box under the SDF tab.



You can access this dialog by invoking the simulator without any arguments or by selecting **Simulate > Simulate** (Main window). See the GUI chapter for a description of this dialog.

For Verilog designs, you can also specify SDF files by using the **$sdf_annotate** system task. See "The $sdf_annotate system task" (UM-294) for more details.

Errors and warnings

Errors issued by the SDF annotator while loading the design prevent the simulation from continuing, whereas warnings do not. Use the **-sdfnoerror** option with **vsim** (CR-189) to change SDF errors to warnings so that the simulation can continue. Warning messages can be suppressed by using **vsim** with either the **-sdfnowarn** or **+nosdfwarn** options.

Another option is to use the **SDF** tab from the **Simulate** dialog box (shown above). Select **Disable SDF warnings** (-sdfnowarn, or +nosdfwarn) to disable warnings, or select **Reduce SDF errors to warnings** (-sdfnoerror) to change errors to warnings.

See "Troubleshooting" (UM-301) for more information on errors and warnings and how to avoid them.

VHDL VITAL SDF

VHDL SDF annotation works on VITAL cells only. The IEEE 1076.4 VITAL ASIC Modeling Specification describes how cells must be written to support SDF annotation. Once again, the designer does not need to know the details of this specification because the library provider has already written the VITAL cells and tools that create compatible SDF files. However, the following summary may help you understand simulator error messages. For additional VITAL specification information, see "VITAL specification and source code" (UM-60).

SDF to VHDL generic matching

An SDF file contains delay and timing constraint data for cell instances in the design. The annotator must locate the cell instances and the placeholders (VHDL generics) for the timing data. Each type of SDF timing construct is mapped to the name of a generic as specified by the VITAL modeling specification. The annotator locates the generic and updates it with the timing value from the SDF file. It is an error if the annotator fails to find the cell instance or the named generic. The following are examples of SDF constructs and their associated generic names:

SDF construct	Matching VHDL generic name
(IOPATH a y (3))	tpd_a_y
(IOPATH (posedge clk) q (1) (2))	tpd_clk_q_posedge
(INTERCONNECT u1/y u2/a (5))	tipd_a
(SETUP d (posedge clk) (5))	tsetup_d_clk_noedge_posedge
(HOLD (negedge d) (posedge clk) (5))	thold_d_clk_negedge_posedge
(SETUPHOLD d clk (5) (5))	tsetup_d_clk & thold_d_clk
(WIDTH (COND (reset==1’b0) clk) (5))	tpw_clk_reset_eq_0

Resolving errors

If the simulator finds the cell instance but not the generic then an error message is issued. For example,

```
** Error (vsim-SDF-3240) myasic.sdf(18):
Instance '/testbench/dut/u1' does not have a generic named 'tpd_a_y'
```

In this case, make sure that the design is using the appropriate VITAL library cells. If it is, then there is probably a mismatch between the SDF and the VITAL cells. You need to find the cell instance and compare its generic names to those expected by the annotator. Look in the VHDL source files provided by the cell library vendor.

If none of the generic names look like VITAL timing generic names, then perhaps the VITAL library cells are not being used. If the generic names do look like VITAL timing generic names but don't match the names expected by the annotator, then there are several possibilities:

- The vendor's tools are not conforming to the VITAL specification.
- The SDF file was accidentally applied to the wrong instance. In this case, the simulator also issues other error messages indicating that cell instances in the SDF could not be located in the design.
- The vendor's library and SDF were developed for the older VITAL 2.2b specification. This version uses different name mapping rules. In this case, invoke **vsim** (CR-189) with the **-vital2.2b** option:

  ```
  vsim -vital2.2b -sdfmax /testbench/u1=myasic.sdf testbench
  ```

For more information on resolving errors see "Troubleshooting" (UM-301).

Verilog SDF

Verilog designs can be annotated using either the simulator command-line options or the **$sdf_annotate** system task (also commonly used in other Verilog simulators). The command-line options annotate the design immediately after it is loaded, but before any simulation events take place. The **$sdf_annotate** task annotates the design at the time it is called in the Verilog source code. This provides more flexibility than the command-line options.

The $sdf_annotate system task

The syntax for **$sdf_annotate** is:

Syntax

```
$sdf_annotate
  (["<sdffile>"], [<instance>], ["<config_file>"], ["<log_file>"],
  ["<mtm_spec>"], ["<scale_factor>"], ["<scale_type>"]);
```

Arguments

`"<sdffile>"`
String that specifies the SDF file. Required.

`<instance>`
Hierarchical name of the instance to be annotated. Optional. Defaults to the instance where the $sdf_annotate call is made.

`"<config_file>"`
String that specifies the configuration file. Optional. Currently not supported, this argument is ignored.

`"<log_file>"`
String that specifies the logfile. Optional. Currently not supported, this argument is ignored.

`"<mtm_spec>"`
String that specifies the delay selection. Optional. The allowed strings are "minimum", "typical", "maximum", and "tool_control". Case is ignored and the default is "tool_control". The "tool_control" argument means to use the delay specified on the command line by +mindelays, +typdelays, or +maxdelays (defaults to +typdelays).

`"<scale_factor>"`
String that specifies delay scaling factors. Optional. The format is "<min_mult>:<typ_mult>:<max_mult>". Each multiplier is a real number that is used to scale the corresponding delay in the SDF file.

`"<scale_type>"`
String that overrides the **<mtm_spec>** delay selection. Optional. The **<mtm_spec>** delay selection is always used to select the delay scaling factor, but if a **<scale_type>** is specified, then it will determine the min/typ/max selection from the SDF file. The allowed strings are "from_min", "from_minimum", "from_typ", "from_typical", "from_max", "from_maximum", and "from_mtm". Case is ignored, and the default is "from_mtm", which means to use the **<mtm_spec>** value.

Examples

Optional arguments can be omitted by using commas or by leaving them out if they are at the end of the argument list. For example, to specify only the SDF file and the instance to which it applies:

```
$sdf_annotate("myasic.sdf", testbench.u1);
```

To also specify maximum delay values:

```
$sdf_annotate("myasic.sdf", testbench.u1, , , "maximum");
```

SDF to Verilog construct matching

The annotator matches SDF constructs to corresponding Verilog constructs in the cells. Usually, the cells contain path delays and timing checks within specify blocks. For each SDF construct, the annotator locates the cell instance and updates each specify path delay or timing check that matches. An SDF construct can have multiple matches, in which case each matching specify statement is updated with the SDF timing value. SDF constructs are matched to Verilog constructs as follows:

IOPATH is matched to specify path delays or primitives:

SDF	Verilog
(IOPATH (posedge clk) q (3) (4))	(posedge clk => q) = 0;
(IOPATH a y (3) (4))	buf u1 (y, a);

The IOPATH construct usually annotates path delays. If the module contains no path delays, then all primitives that drive the specified output port are annotated.

INTERCONNECT and **PORT** are matched to input ports:

SDF	Verilog
(INTERCONNECT u1.y u2.a (5))	input a;
(PORT u2.a (5))	inout a;

Both of these constructs identify a module input or inout port and create an internal net that is a delayed version of the port. This is called a Module Input Port Delay (MIPD). All primitives, specify path delays, and specify timing checks connected to the original port are reconnected to the new MIPD net.

PATHPULSE and **GLOBALPATHPULSE** are matched to specify path delays:

SDF	Verilog
(PATHPULSE a y (5) (10))	(a => y) = 0;
(GLOBALPATHPULSE a y (30) (60))	(a => y) = 0;

If the input and output ports are omitted in the SDF, then all path delays are matched in the cell.

DEVICE is matched to primitives or specify path delays:

SDF	Verilog
(DEVICE y (5))	and u1(y, a, b);
(DEVICE y (5))	(a => y) = 0; (b => y) = 0;

If the SDF cell instance is a primitive instance, then that primitive's delay is annotated. If it is a module instance, then all specify path delays are annotated that drive the output port specified in the DEVICE construct (all path delays are annotated if the output port is omitted). If the module contains no path delays, then all primitives that drive the specified output port are annotated (or all primitives that drive any output port if the output port is omitted).

SETUP is matched to $setup and $setuphold:

SDF	Verilog
(SETUP d (posedge clk) (5))	$setup(d, posedge clk, 0);
(SETUP d (posedge clk) (5))	$setuphold(posedge clk, d, 0, 0);

HOLD is matched to $hold and $setuphold:

SDF	Verilog
(HOLD d (posedge clk) (5))	$hold(posedge clk, d, 0);
(HOLD d (posedge clk) (5))	$setuphold(posedge clk, d, 0, 0);

SETUPHOLD is matched to $setup, $hold, and $setuphold:

SDF	Verilog
(SETUPHOLD d (posedge clk) (5) (5))	$setup(d, posedge clk, 0);
(SETUPHOLD d (posedge clk) (5) (5))	$hold(posedge clk, d, 0);
(SETUPHOLD d (posedge clk) (5) (5))	$setuphold(posedge clk, d, 0, 0);

RECOVERY is matched to $recovery:

SDF	Verilog
(RECOVERY (negedge reset) (posedge clk) (5))	$recovery(negedge reset, posedge clk, 0);

REMOVAL is matched to $removal:

SDF	Verilog
(REMOVAL (negedge reset) (posedge clk) (5))	$removal(negedge reset, posedge clk, 0);

RECREM is matched to $recovery, $removal, and $recrem:

SDF	Verilog
(RECREM (negedge reset) (posedge clk) (5) (5))	$recovery(negedge reset, posedge clk, 0);
(RECREM (negedge reset) (posedge clk) (5) (5))	$removal(negedge reset, posedge clk, 0);
(RECREM (negedge reset) (posedge clk) (5) (5))	$recrem(negedge reset, posedge clk, 0);

SKEW is matched to $skew:

SDF	Verilog
(SKEW (posedge clk1) (posedge clk2) (5))	$skew(posedge clk1, posedge clk2, 0);

WIDTH is matched to $width:

SDF	Verilog
(WIDTH (posedge clk) (5))	$width(posedge clk, 0);

PERIOD is matched to $period:

SDF	Verilog
(PERIOD (posedge clk) (5))	$period(posedge clk, 0);

NOCHANGE is matched to $nochange:

SDF	Verilog
(NOCHANGE (negedge write) addr (5) (5))	$nochange(negedge write, addr, 0, 0);

Optional edge specifications

Timing check ports and path delay input ports can have optional edge specifications. The annotator uses the following rules to match edges:

- A match occurs if the SDF port does not have an edge.
- A match occurs if the specify port does not have an edge.
- A match occurs if the SDF port edge is identical to the specify port edge.
- A match occurs if explicit edge transitions in the specify port edge overlap with the SDF port edge.

These rules allow SDF annotation to take place even if there is a difference between the number of edge-specific constructs in the SDF file and the Verilog specify block. For example, the Verilog specify block may contain separate setup timing checks for a falling and rising edge on data with respect to clock, while the SDF file may contain only a single setup check for both edges:

SDF	Verilog
(SETUP data (posedge clock) (5))	$setup(posedge data, posedge clk, 0);
(SETUP data (posedge clock) (5))	$setup(negedge data, posedge clk, 0);

In this case, the cell accommodates more accurate data than can be supplied by the tool that created the SDF file, and both timing checks correctly receive the same value. Likewise, the SDF file may contain more accurate data than the model can accommodate.

SDF	Verilog
(SETUP (posedge data) (posedge clock) (4))	$setup(data, posedge clk, 0);
(SETUP (negedge data) (posedge clock) (6))	$setup(data, posedge clk, 0);

In this case, both SDF constructs are matched and the timing check receives the value from the last one encountered.

Timing check edge specifiers can also use explicit edge transitions instead of posedge and negedge. However, the SDF file is limited to posedge and negedge. The explicit edge specifiers are 01, 0x, 10, 1x, x0, and x1. The set of [01, 0x, x1] is equivalent to posedge, while the set of [10, 1x, x0] is equivalent to negedge. A match occurs if any of the explicit edges in the specify port match any of the explicit edges implied by the SDF port. For example,

SDF	Verilog
(SETUP data (posedge clock) (5))	$setup(data, edge[01, 0x] clk, 0);

Optional conditions

Timing check ports and path delays can have optional conditions. The annotator uses the following rules to match conditions:

- A match occurs if the SDF does not have a condition.
- A match occurs for a timing check if the SDF port condition is semantically equivalent to the specify port condition.
- A match occurs for a path delay if the SDF condition is lexically identical to the specify condition.

Timing check conditions are limited to very simple conditions, therefore the annotator can match the expressions based on semantics. For example,

SDF	Verilog
(SETUP data (COND (reset!=1) (posedge clock)) (5))	$setup(data, posedge clk &&& (reset==0), 0);

The conditions are semantically equivalent and a match occurs. In contrast, path delay conditions may be complicated and semantically equivalent conditions may not match. For example,

SDF	Verilog
(COND (r1 || r2) (IOPATH clk q (5)))	if (r1 || r2) (clk => q) = 5; // matches
(COND (r1 || r2) (IOPATH clk q (5)))	if (r2 || r1) (clk => q) = 5; // does not match

The annotator does not match the second condition above because the order of r1 and r2 are reversed.

Rounded timing values

The SDF **TIMESCALE** construct specifies time units of values in the SDF file. The annotator rounds timing values from the SDF file to the time precision of the module that is annotated. For example, if the SDF TIMESCALE is 1ns and a value of .016 is annotated to a path delay in a module having a time precision of 10ps (from the timescale directive), then the path delay receives a value of 20ps. The SDF value of 16ps is rounded to 20ps. Interconnect delays are rounded to the time precision of the module that contains the annotated MIPD.

SDF for Mixed VHDL and Verilog Designs

Annotation of a mixed VHDL and Verilog design is very flexible. VHDL VITAL cells and Verilog cells can be annotated from the same SDF file. This flexibility is available only by using the simulator's SDF command-line options. The Verilog $sdf_annotate system task can annotate Verilog cells only. See the **vsim** command (CR-189) for more information on SDF command-line options.

Interconnect delays

An interconnect delay represents the delay from the output of one device to the input of another. ModelSim can model single interconnect delays or multisource interconnect delays for Verilog, VHDL/VITAL, or mixed designs. See the **vsim** command for more information on the relevant command-line arguments.

Timing checks are performed on the interconnect delayed versions of input ports. This may result in misleading timing constraint violations, because the ports may satisfy the constraint while the delayed versions may not. If the simulator seems to report incorrect violations, be sure to account for the effect of interconnect delays.

Disabling timing checks

ModelSim offers a number of options for disabling timing checks on a "global" or individual basis. The table below provides a summary of those options. See the command and argument descriptions in the *ModelSim Command Reference* for more details.

Command and argument	Effect
vlog +notimingchecks	disables timing check system tasks for all instances in the specified Verilog design
vlog +nospecify	disables specify path delays and timing checks for all instances in the specified Verilog design
vsim +no_neg_tchk	disables negative timing check limits by setting them to zero for all instances in the specified design
vsim +no_notifier	disables the toggling of the notifier register argument of the timing check system tasks for all instances in the specified design
vsim +no_tchk_msg	disables error messages issued by timing check system tasks when timing check violations occur for all instances in the specified design
vsim +notimingchecks	disables Verilog and VITAL timing checks for all instances in the specified design

Troubleshooting

Specifying the wrong instance

By far, the most common mistake in SDF annotation is to specify the wrong instance to the simulator's SDF options. The most common case is to leave off the instance altogether, which is the same as selecting the top-level design unit. This is generally wrong because the instance paths in the SDF are relative to the ASIC or FPGA model, which is usually instantiated under a top-level testbench. See "Instance specification" (UM-290) for an example.

A common example for both VHDL and Verilog test benches is provided below. For simplicity, the test benches do nothing more than instantiate a model that has no ports.

VHDL testbench

```
entity testbench is end;

architecture only of testbench is
    component myasic
    end component;
begin
    dut : myasic;
end;
```

Verilog testbench

```
module testbench;
    myasic dut();
endmodule
```

The name of the model is *myasic* and the instance label is *dut*. For either testbench, an appropriate simulator invocation might be:

```
vsim -sdfmax /testbench/dut=myasic.sdf testbench
```

Optionally, you can leave off the name of the top-level:

```
vsim -sdfmax /dut=myasic.sdf testbench
```

The important thing is to select the instance for which the SDF is intended. If the model is deep within the design hierarchy, an easy way to find the instance name is to first invoke the simulator without SDF options, open the structure window, navigate to the model instance, select it, and enter the **environment** command (CR-74). This command displays the instance name that should be used in the SDF command-line option.

Mistaking a component or module name for an instance label

Another common error is to specify the component or module name rather than the instance label. For example, the following invocation is wrong for the above testbenches:

```
vsim -sdfmax /testbench/myasic=myasic.sdf testbench
```

This results in the following error message:

```
** Error (vsim-SDF-3250) myasic.sdf(0):
Failed to find INSTANCE '/testbench/myasic'.
```

Forgetting to specify the instance

If you leave off the instance altogether, then the simulator issues a message for each instance path in the SDF that is not found in the design. For example,

```
vsim -sdfmax myasic.sdf testbench
```

Results in:

```
** Error (vsim-SDF-3250) myasic.sdf(0):
Failed to find INSTANCE '/testbench/u1'
** Error (vsim-SDF-3250) myasic.sdf(0):
Failed to find INSTANCE '/testbench/u2'
** Error (vsim-SDF-3250) myasic.sdf(0):
Failed to find INSTANCE '/testbench/u3'
** Error (vsim-SDF-3250) myasic.sdf(0):
Failed to find INSTANCE '/testbench/u4'
** Error (vsim-SDF-3250) myasic.sdf(0):
Failed to find INSTANCE '/testbench/u5'
** Warning (vsim-SDF-3432) myasic.sdf:
This file is probably applied to the wrong instance.
** Warning (vsim-SDF-3432) myasic.sdf:
Ignoring subsequent missing instances from this file.
```

After annotation is done, the simulator issues a summary of how many instances were not found and possibly a suggestion for a qualifying instance:

```
** Warning (vsim-SDF-3440) myasic.sdf:
Failed to find any of the 358 instances from this file.
** Warning (vsim-SDF-3442) myasic.sdf:
Try instance '/testbench/dut'. It contains all instance paths from this
file.
```

The simulator recommends an instance only if the file was applied to the top-level and a qualifying instance is found one level down.

Also see "Resolving errors" (UM-293) for specific VHDL VITAL SDF troubleshooting.

10 - Value Change Dump (VCD) Files

Chapter contents

ModelSim VCD commands and VCD tasks UM-304

Creating a VCD file UM-306
Flow for four-state VCD file UM-306
Flow for extended VCD file UM-306

Resimulating a design from a VCD file UM-307
Example 1 — Verilog counter UM-307
Example 2 — VHDL adder UM-307
Example 3 — Mixed-HDL design UM-308

A VCD file from source to output UM-309
VHDL source code UM-309
VCD simulator commands UM-309
VCD output UM-310

Capturing port driver data UM-312
Supported TSSI states UM-312
Strength values UM-313
Port identifier code UM-313
Example VCD output from vcd dumpports UM-314

This chapter explains Model Technology's Verilog VCD implementation for ModelSim.

The VCD file format is specified in the IEEE 1364 standard. It is an ASCII file containing header information, variable definitions, and variable value changes. VCD is in common use for Verilog designs, and is controlled by VCD system task calls in the Verilog source code. ModelSim provides simulator command equivalents for these system tasks and extends VCD support to VHDL designs; the ModelSim commands can be used on either VHDL or Verilog designs.

▶ **Note:** If you need vendor-specific ASIC design-flow documentation that incorporates VCD, please contact your ASIC vendor.

ModelSim VCD commands and VCD tasks

ModelSim VCD commands map to IEEE Std 1364 VCD system tasks and appear in the VCD file along with the results of those commands. The table below maps the VCD commands to their associated tasks.

VCD commands	VCD system tasks
vcd add (CR-127)	$dumpvars
vcd checkpoint (CR-128)	$dumpall
vcd file (CR-136)▲	$dumpfile
vcd flush (CR-140)	$dumpflush
vcd limit (CR-141)	$dumplimit
vcd off (CR-142)	$dumpoff
vcd on (CR-143)	$dumpon

ModelSim versions 5.5 and later also support extended VCD (dumpports system tasks). The table below maps the VCD dumpports commands to their associated tasks.

VCD dumpports commands	VCD system tasks
vcd dumpports (CR-130)	$dumpports
vcd dumpportsall (CR-131)	$dumpportsall
vcd dumpportsflush (CR-132)	$dumpportsflush
vcd dumpportslimit (CR-133)	$dumpportslimit
vcd dumpportsoff (CR-134)	$dumpportsoff
vcd dumpportson (CR-135)	$dumpportson

ModelSim versions 5.5 and later support multiple VCD files. This functionality is an extension of the IEEE Std 1364 specification. The tasks behave the same as the IEEE equivalent tasks such as $dumpfile, $dumpvar, etc. The difference is that $fdumpfile can be called multiple times to create more than one VCD file, and the remaining tasks require a filename argument to associate their actions with a specific file.

VCD commands	VCD system tasks
vcd add (CR-127) `-file <filename>`	$fdumpvars
vcd checkpoint (CR-128) `<filename>`	$fdumpall
vcd files (CR-138) `<filename>`▲	$fdumpfile
vcd flush (CR-140) `<filename>`	$fdumpflush

VCD commands	VCD system tasks
vcd limit (CR-141) `<filename>`	$fdumplimit
vcd off (CR-142) `<filename>`	$fdumpoff
vcd on (CR-143) `<filename>`	$fdumpon

Important: Note that two commands (**vcd file** and **vcd files**) are available to specify a filename and state mapping for a VCD file. **Vcd file** allows for only one VCD file and exists for backwards compatibility with ModelSim versions prior to 5.5. **Vcd files** allows for creation of multiple VCD files and is the preferred command to use in ModelSim versions 5.5 and later.

Creating a VCD file

There are two flows in ModelSim for creating a VCD file. One flow produces a four-state VCD file with variable changes in 0, 1, x, and z with no strength information; the other produces an extended VCD file with variable changes in all states and strength information and port driver data.

Both flows will also capture port driver changes unless filtered out with optional command-line arguments.

The commands shown below are documented in detail in the *ModelSim Command Reference*.

Flow for four-state VCD file

First, compile and load the design:

```
% cd ~/modeltech/examples
% vlib work
% vlog counter.v tcounter.v
% vsim test_counter
```

Next, with the design loaded, specify the VCD file name with the **vcd file** command (CR-136) and add items to the file with the **vcd add** command (CR-127):

```
VSIM 1> vcd file myvcdfile.vcd
VSIM 2> vcd add /test_counter/dut/*
VSIM 3> run
VSIM 4> quit -f
```

There will now be a VCD file in the working directory.

Flow for extended VCD file

First, compile and load the design:

```
% cd ~/modeltech/examples
% vlib work
% vlog counter.v tcounter.v
% vsim test_counter
```

Next, with the design loaded, specify the VCD file name and items to add with the **vcd dumpports** command (CR-130):

```
VSIM 1> vcd dumpports -file myvcdfile.vcd /test_counter/dut/*
VSIM 3> run
VSIM 4> quit -f
```

There will now be an extended VCD file in the working directory.

Case sensitivity

VHDL is not case sensitive so ModelSim converts all signal names to lower case when it produces a VCD file. Conversely, Verilog designs are case sensitive so ModelSim maintains case when it produces a VCD file.

Resimulating a design from a VCD file

To resimulate with a VCD file, you capture the ports of a design unit instance within a testbench or design. The design may be VHDL, Verilog, or mixed HDL. You can resimulate only at the top level of the module for which you captured ports.

The general procedure for resimulating with a VCD file includes two steps:

1. Create a VCD file using the **vcd dumpports** command (CR-130).
2. Rerun without the testbench, using the **-vcdstim** argument to **vsim** (CR-189). Note that **-vcdstim** works only with VCD files that were created by a ModelSim simulation.

Example 1 — Verilog counter

First, create the VCD file using **vcd dumpports**:

```
% cd ~/modeltech/examples
% vlib work
% vlog counter.v tcounter.v
% vsim test_counter
VSIM 1> vcd dumpports -file counter.vcd /test_counter/dut/*
VSIM 2> run
VSIM 3> quit -f
```

Next, rerun the counter without the testbench, using the **-vcdstim** argument:

```
% vsim -vcdstim counter.vcd counter
VSIM 1> add wave /*
VSIM 2> run 200
```

Example 2 — VHDL adder

First, create the VCD file using **vcd dumpports**:

```
% cd ~/modeltech/examples
% vlib work
% vcom gates.vhd adder.vhd stimulus.vhd
% vsim testbench
VSIM 1> vcd dumpports -file addern.vcd /testbench/uut/*
VSIM 2> run 1000
VSIM 3> quit -f
```

Next, rerun the adder without the testbench, using the **-vcdstim** argument:

```
% vsim -vcdstim addern.vcd addern -gn=8 -do "add wave /*; run 1000"
```

Example 3 — Mixed-HDL design

First, create three VCD files, one for each module:

```
% cd ~/modeltech/examples/mixedHDL
% vlib work
% vlog cache.v memory.v proc.v
% vcom util.vhd set.vhd top.vhd
% vsim top
VSIM 1> vcd dumpports -file proc.vcd /top/p/*
VSIM 2> vcd dumpports -file cache.vcd /top/c/*
VSIM 3> vcd dumpports -file memory.vcd /top/m/*
VSIM 4> run 1000
VSIM 5> quit -f
```

Next, rerun each module separately, using the captured VCD stimulus:

```
% vsim -vcdstim proc.vcd proc -do "add wave /*; run 1000"
VSIM 1> quit -f

% vsim -vcdstim cache.vcd cache -do "add wave /*; run 1000"
VSIM 1> quit -f

% vsim -vcdstim memory.vcd memory -do "add wave /*; run 1000"
VSIM 1> quit -f
```

A VCD file from source to output

The following example shows the VHDL source, a set of simulator commands, and the resulting VCD output.

VHDL source code

The design is a simple shifter device represented by the following VHDL source code:

```
library IEEE;
use IEEE.STD_LOGIC_1164.all;

entity SHIFTER_MOD is
    port (CLK, RESET, data_in   : IN STD_LOGIC;
        Q : INOUT STD_LOGIC_VECTOR(8 downto 0));
END SHIFTER_MOD ;

architecture RTL of SHIFTER_MOD is
begin
    process (CLK,RESET)
    begin
        if (RESET = '1') then
            Q <= (others => '0') ;
        elsif (CLK'event and CLK = '1') then
            Q <= Q(Q'left - 1 downto 0) & data_in ;
        end if ;
    end process ;
end ;
```

VCD simulator commands

At simulator time zero, the designer executes the following commands and quits the simulator at time 1200:

```
vcd file output.vcd
vcd add -r *
force reset 1 0
force data_in 0 0
force clk 0 0
run 100
force clk 1 0, 0 50 -repeat 100
run 100
vcd off
force reset 0 0
force data_in 1 0
run 100
vcd on
run 850
force reset 1 0
run 50
vcd checkpoint
```

VCD output

The VCD file created as a result of the preceding scenario would be called *output.vcd.* The following pages show how it would look.

VCD output

```
$comment
    File created using the following
command:
        vcd files output.vcd
$date
    Fri Jan 12 09:07:17 2000
$end
$version
    ModelSim EE/PLUS 5.4
$end
$timescale
    1ns
$end
$scope module shifter_mod $end
$var wire 1 ! clk $end
$var wire 1 " reset $end
$var wire 1 # data_in $end
$var wire 1 $ q [8] $end
$var wire 1 % q [7] $end
$var wire 1 & q [6] $end
$var wire 1 ’ q [5] $end
$var wire 1 ( q [4] $end
$var wire 1 ) q [3] $end
$var wire 1 * q [2] $end
$var wire 1 + q [1] $end
$var wire 1 , q [0] $end
$upscope $end
$enddefinitions $end
#0
$dumpvars
0!
1"
0#
0$
0%
0&
0’
0(
0)
0*
0+
0,
$end
#100
1!
#150
0!
#200
1!
$dumpoff
x!
x"
x#
x$
x%
x&
x’
x(
x)
x*
x+
x,
$end
#300
$dumpon
1!
0"
1#
0$
0%
```

```
0&
0’
0(
0)
0*
0+
1,
$end
#350
0!
#400
1!
1+
#450
0!
#500
1!
1*
#550
0!
#600
1!
1)
#650
0!
#700
1!
1(
#750
0!
#800
1!
1’
#850
0!
#900
1!
1&
#950
0!
```

```
#1000
1!
1%
#1050
0!
#1100
1!
1$
#1150
0!
1"
0$
0%
0&
0’
0(
0)
0*
0+
0,
#1200
1!
$dumpall
1!
1"
1#
0$
0%
0&
0’
0(
0)
0*
0+
0,
$end
```

Capturing port driver data

Some ASIC vendors' toolkits read a VCD file format that provides details on port drivers. This information can be used, for example, to drive a tester. See the ASIC vendor's documentation for toolkit specific information.

In ModelSim use the **vcd dumpports** command (CR-130) to create a VCD file that captures port driver data.

Port driver direction information is captured as TSSI states in the VCD file. Each time an external or internal port driver changes values, a new value change is recorded in the VCD file with the following format:

```
p<TSSI state> <0 strength> <1 strength> <identifier_code>
```

Supported TSSI states

The supported <TSSI states> are:

Input (testfixture)	Output (dut)
D low	L low
U high	H high
N unknown	X unknown
Z tri-state	T tri-state

Unknown direction
0 low (both input and output are driving low)
1 high (both input and output are driving high)
? unknown (both input and output are driving unknown)
f tri-state
A unknown (input driving low and output driving high)
a unknown (input driving low and output driving unknown)
C unknown (input driving unknown and output driving low)
b unknown (input driving high and output driving unknown)
B unknown (input driving high and output driving low)
c unknown (input driving unknown and output driving high)

Strength values

The <strength> values are based on Verilog strengths:

Strength	VHDL std_logic mappings
0 highz	’Z’
1 small	
2 medium	
3 weak	
4 large	
5 pull	’W’,’H’,’L’
6 strong	’U’,’X’,’0’,’1’,’-’
7 supply	

Port identifier code

The <identifier_code> is an integer preceded by < that starts at zero and is incremented for each port in the order the ports are specified. Also, the variable type recorded in the VCD header is "port".

Example VCD output from vcd dumpports

The following is an example VCD file created with the **vcd dumpports** command.

```
$comment
File created using the following command:
vcd dumpports results/dump1
$end
$date
Tue Aug 20 13:33:02 2000
$end
$version
ModelSim Version 5.4c
$end
$timescale
1ns
$end
$scope module top1 $end
$scope module u1 $end
$var port 1 <0 a $end
$var port 1 <1 b $end
$var port 1 <2 c $end
$upscope $end
$upscope $end
$enddefinitions $end
#0
$dumpports
pN 6 6 <0
pX 6 6 <1
p? 6 6 <2
$end
#10
pX 6 6 <1
pN 6 6 <0
p? 6 6 <2
#20
pL 6 0 <1
pD 6 0 <0
pa 6 6 <2
#30
pH 0 6 <1
pU 0 6 <0
pb 6 6 <2
#40
pT 0 0 <1
pZ 0 0 <0
pX 6 6 <2
#50
pX 5 5 <1
pN 5 5 <0
p? 6 6 <2
#60
pL 5 0 <1
pD 5 0 <0
pa 6 6 <2
#70
pH 0 5 <1
pU 0 5 <0
pb 6 6 <2
#80
pX 6 6 <1
pN 6 6 <0
p? 6 6 <2
```

11 - Tcl and macros (DO files)

Chapter contents

Tcl features within ModelSim UM-316

Tcl References UM-316

Tcl commands UM-317

Tcl command syntax UM-318
- if command syntax UM-320
- set command syntax UM-321
- Command substitution UM-322
- Command separator UM-322
- Multiple-line commands UM-322
- Evaluation order UM-322
- Tcl relational expression evaluation UM-322
- Variable substitution UM-323
- System commands. UM-323

List processing UM-324

ModelSim Tcl commands UM-324

ModelSim Tcl time commands UM-325

Tcl examples UM-327

Macros (DO files) UM-331
- Creating DO files UM-331
- Using Parameters with DO files UM-331
- Making macro parameters optional UM-332
- Useful commands for handling breakpoints and errors . . . UM-333
- Error action in DO files UM-333

This chapter provides an overview of Tcl (tool command language) as used with ModelSim. Macros in ModelSim are simply Tcl scripts that contain ModelSim and, optionally, Tcl commands.

Tcl is a scripting language for controlling and extending ModelSim. Within ModelSim you can develop implementations from Tcl scripts without the use of C code. Because Tcl is interpreted, development is rapid; you can generate and execute Tcl scripts on the fly without stopping to recompile or restart ModelSim. In addition, if ModelSim does not provide the command you need, you can use Tcl to create your own commands.

Tcl features within ModelSim

Using Tcl with ModelSim gives you these features:

- command history (like that in C shells)
- full expression evaluation and support for all C-language operators
- a full range of math and trig functions
- support of lists and arrays
- regular expression pattern matching
- procedures
- the ability to define your own commands
- command substitution (that is, commands may be nested)
- robust scripting language for macros

Tcl References

Two books about Tcl are *Tcl and the Tk Toolkit* by John K. Ousterhout, published by Addison-Wesley Publishing Company, Inc., and *Practical Programming in Tcl and Tk by* Brent Welch published by Prentice Hall. You can also consult the following online references:

- Select **Help > Tcl Man Pages** (Main window).
- The Model Technology web site lists a variety of Tcl resources: www.model.com/resources/tcltk.asp

Tcl commands

For complete information on Tcl commands, select **Help > Tcl Man Pages** (Main window). Also see "Preference variables located in Tcl files" (UM-352) for information on Tcl variables.

ModelSim command names that conflict with Tcl commands have been renamed or have been replaced by Tcl commands. See the list below:

Previous ModelSim command	Command changed to (or replaced by)
continue	**run** (CR-114) with the **-continue** option
format list | wave	**write format** (CR-216) with either list or wave specified
if	replaced by the Tcl **if** command, see "if command syntax" (UM-320) for more information
list	**add list** (CR-32)
nolist | nowave	**delete** (CR-65) with either list or wave specified
set	replaced by the Tcl **set** command, see "set command syntax" (UM-321) for more information
source	**vsource** (CR-204)
wave	**add wave** (CR-35)

Tcl command syntax

The following eleven rules define the syntax and semantics of the Tcl language. Additional details on if command syntax (UM-320) and set command syntax (UM-321) follow.

1 A Tcl script is a string containing one or more commands. Semi-colons and newlines are command separators unless quoted as described below. Close brackets ("]") are command terminators during command substitution (see below) unless quoted.

2 A command is evaluated in two steps. First, the Tcl interpreter breaks the command into words and performs substitutions as described below. These substitutions are performed in the same way for all commands. The first word is used to locate a command procedure to carry out the command, then all of the words of the command are passed to the command procedure. The command procedure is free to interpret each of its words in any way it likes, such as an integer, variable name, list, or Tcl script. Different commands interpret their words differently.

3 Words of a command are separated by white space (except for newlines, which are command separators).

4 If the first character of a word is double-quote (""") then the word is terminated by the next double-quote character. If semi-colons, close brackets, or white space characters (including newlines) appear between the quotes then they are treated as ordinary characters and included in the word. Command substitution, variable substitution, and backslash substitution are performed on the characters between the quotes as described below. The double-quotes are not retained as part of the word.

5 If the first character of a word is an open brace ("{") then the word is terminated by the matching close brace ("}"). Braces nest within the word: for each additional open brace there must be an additional close brace (however, if an open brace or close brace within the word is quoted with a backslash then it is not counted in locating the matching close brace). No substitutions are performed on the characters between the braces except for backslash-newline substitutions described below, nor do semi-colons, newlines, close brackets, or white space receive any special interpretation. The word will consist of exactly the characters between the outer braces, not including the braces themselves.

6 If a word contains an open bracket ("[") then Tcl performs command substitution. To do this it invokes the Tcl interpreter recursively to process the characters following the open bracket as a Tcl script. The script may contain any number of commands and must be terminated by a close bracket ("]"). The result of the script (i.e. the result of its last command) is substituted into the word in place of the brackets and all of the characters between them. There may be any number of command substitutions in a single word. Command substitution is not performed on words enclosed in braces.

7 If a word contains a dollar-sign ("$") then Tcl performs variable substitution: the dollar-sign and the following characters are replaced in the word by the value of a variable. Variable substitution may take any of the following forms:

```
$name
```

Name is the name of a scalar variable; the name is terminated by any character that isn't a letter, digit, or underscore.

```
$name(index)
```

Name gives the name of an array variable and index gives the name of an element within that array. Name must contain only letters, digits, and underscores. Command substitutions, variable substitutions, and backslash substitutions are performed on the characters of index.

```
${name}
```

Name is the name of a scalar variable. It may contain any characters whatsoever except for close braces.

There may be any number of variable substitutions in a single word. Variable substitution is not performed on words enclosed in braces.

8 If a backslash ("\") appears within a word then backslash substitution occurs. In all cases but those described below the backslash is dropped and the following character is treated as an ordinary character and included in the word. This allows characters such as double quotes, close brackets, and dollar signs to be included in words without triggering special processing. The following table lists the backslash sequences that are handled specially, along with the value that replaces each sequence.

`\a`	Audible alert (bell) (0x7).
`\b`	Backspace (0x8).
`\f`	Form feed (0xc).
`\n`	Newline (0xa).
`\r`	Carriage-return (0xd).
`\t`	Tab (0x9).
`\v`	Vertical tab (0xb).
`\<newline>whiteSpace`	A single space character replaces the backslash, newline, and all spaces and tabs after the newline. This backslash sequence is unique in that it is replaced in a separate pre-pass before the command is actually parsed. This means that it will be replaced even when it occurs between braces, and the resulting space will be treated as a word separator if it isn't in braces or quotes.
`\\`	Backslash ("\").
`\`*ooo*	The digits ooo (one, two, or three of them) give the octal value of the character.

\x*hh*	The hexadecimal digits hh give the hexadecimal value of the character. Any number of digits may be present.

Backslash substitution is not performed on words enclosed in braces, except for backslash-newline as described above.

9 If a hash character ("#") appears at a point where Tcl is expecting the first character of the first word of a command, then the hash character and the characters that follow it, up through the next newline, are treated as a comment and ignored. The comment character only has significance when it appears at the beginning of a command.

10 Each character is processed exactly once by the Tcl interpreter as part of creating the words of a command. For example, if variable substitution occurs then no further substitutions are performed on the value of the variable; the value is inserted into the word verbatim. If command substitution occurs then the nested command is processed entirely by the recursive call to the Tcl interpreter; no substitutions are performed before making the recursive call and no additional substitutions are performed on the result of the nested script.

11 Substitutions do not affect the word boundaries of a command. For example, during variable substitution the entire value of the variable becomes part of a single word, even if the variable's value contains spaces.

if command syntax

The Tcl **if** command executes scripts conditionally. Note that in the syntax below the "?" indicates an optional argument.

Syntax

```
if expr1 ?then? body1 elseif expr2 ?then? body2 elseif ... ?else? ?bodyN?
```

Description

The **if** command evaluates *expr1* as an expression. The value of the expression must be a boolean (a numeric value, where 0 is false and anything else is true, or a string value such as **true** or **yes** for true and **false** or **no** for false); if it is true then *body1* is executed by passing it to the Tcl interpreter. Otherwise *expr2* is evaluated as an expression and if it is true then *body2* is executed, and so on. If none of the expressions evaluates to true then *bodyN* is executed. The **then** and **else** arguments are optional "noise words" to make the command easier to read. There may be any number of **elseif** clauses, including zero. *BodyN* may also be omitted as long as **else** is omitted too. The return value from the command is the result of the body script that was executed, or an empty string if none of the expressions was non-zero and there was no *bodyN*.

set command syntax

The Tcl **set** command reads and writes variables. Note that in the syntax below the "?" indicates an optional argument.

Syntax

```
set varName ?value?
```

Description

Returns the value of variable *varName*. If *value* is specified, then sets the value of *varName* to value, creating a new variable if one doesn't already exist, and returns its value. If *varName* contains an open parenthesis and ends with a close parenthesis, then it refers to an array element: the characters before the first open parenthesis are the name of the array, and the characters between the parentheses are the index within the array. Otherwise *varName* refers to a scalar variable. Normally, *varName* is unqualified (does not include the names of any containing namespaces), and the variable of that name in the current namespace is read or written. If *varName* includes namespace qualifiers (in the array name if it refers to an array element), the variable in the specified namespace is read or written.

If no procedure is active, then *varName* refers to a namespace variable (global variable if the current namespace is the global namespace). If a procedure is active, then *varName* refers to a parameter or local variable of the procedure unless the global command was invoked to declare *varName* to be global, or unless a Tcl **variable** command was invoked to declare varName to be a namespace variable.

Command substitution

Placing a command in square brackets [ ] will cause that command to be evaluated first and its results returned in place of the command. An example is:

```
set a 25
set b 11
set c 3
echo "the result is [expr ($a + $b)/$c]"
```

will output:

```
"the result is 12"
```

This feature allows VHDL variables and signals, and Verilog nets and registers to be accessed using:

```
[examine -<radix> name]
```

The %name substitution is no longer supported. Everywhere %name could be used, you now can use [examine -value -<radix> name] which allows the flexibility of specifying command options. The radix specification is optional.

Command separator

A semicolon character (;) works as a separator for multiple commands on the same line. It is not required at the end of a line in a command sequence.

Multiple-line commands

With Tcl, multiple-line commands can be used within macros and on the command line. The command line prompt will change (as in a C shell) until the multiple-line command is complete.

In the example below, note the way the opening brace '{' is at the end of the if and else lines. This is important because otherwise the Tcl scanner won't know that there is more coming in the command and will try to execute what it has up to that point, which won't be what you intend.

```
if { [exa sig_a] == "0011ZZ"}  {
    echo "Signal value matches"
    do macro_1.do
} else {
    echo "Signal value fails"
    do macro_2.do  }
```

Evaluation order

An important thing to remember when using Tcl is that anything put in curly brackets {} is not evaluated immediately. This is important for if-then-else, procedures, loops, and so forth.

Tcl relational expression evaluation

When you are comparing values, the following hints may be useful:

- Tcl stores all values as strings, and will convert certain strings to numeric values when appropriate. If you want a literal to be treated as a numeric value, don't quote it.

  ```
  if {[exa var_1] == 345}...
  ```

  The following will also work:

  ```
  if {[exa var_1] == "345"}...
  ```

- However, if a literal cannot be represented as a number, you *must* quote it, or Tcl will give you an error. For instance:

  ```
  if {[exa var_2] == 001Z}...
  ```

  will give an error.

  ```
  if {[exa var_2] == "001Z"}...
  ```

  will work okay.

- Don't quote single characters in single quotes:

  ```
  if {[exa var_3] == 'X'}...
  ```

  will give an error

  ```
  if {[exa var_3] == "X"}...
  ```

  will work okay.

- For the equal operator, you must use the C operator "==" . For not-equal, you must use the C operator "!=".

Variable substitution

When a $<var_name> is encountered, the Tcl parser will look for variables that have been defined either by ModelSim or by you, and substitute the value of the variable.

Note: Tcl is case sensitive for variable names.

To access environment variables, use the construct:

```
$env(<var_name>)
echo My user name is $env(USER)
```

Environment variables can also be set using the env array:

```
set env(SHELL) /bin/csh
```

See "Simulator state variables" (UM-353) for more information about ModelSim-defined variables.

System commands

To pass commands to the DOS window, use the Tcl **exec** command:

```
echo The date is [exec date]
```

List processing

In Tcl a "list" is a set of strings in curly braces separated by spaces. Several Tcl commands are available for creating lists, indexing into lists, appending to lists, getting the length of lists and shifting lists. These commands are:

Command syntax	Description
lappend var_name val1 val2 ...	appends val1, val2, etc. to list var_name
lindex list_name index	returns the index-th element of list_name; the first element is 0
linsert list_name index val1 val2 ...	inserts val1, val2, etc. just before the index-th element of list_name
list val1, val2 ...	returns a Tcl list consisting of val1, val2, etc.
llength list_name	returns the number of elements in list_name
lrange list_name first last	returns a sublist of list_name, from index first to index last; first or last may be "end", which refers to the last element in the list
lreplace list_name first last val1, val2, ...	replaces elements first through last with val1, val2, etc.

Two other commands, **lsearch** and **lsort,** are also available for list manipulation. See the Tcl man pages (**Help > Tcl Man Pages**) for more information on these commands.

ModelSim Tcl commands

These additional commands enhance the interface between Tcl and ModelSim. Only brief descriptions are provided here; for more information and command syntax see the *ModelSim Command Reference*.

Command	Description
alias (CR-39)	creates a new Tcl procedure that evaluates the specified commands; used to create a user-defined alias
find (CR-79)	locates incrTcl classes and objects
lshift (CR-89)	takes a Tcl list as argument and shifts it in-place one place to the left, eliminating the 0th element
lsublist (CR-90)	returns a sublist of the specified Tcl list that matches the specified Tcl glob pattern
printenv (CR-103)	echoes to the Main window the current names and values of all environment variables

ModelSim Tcl time commands

ModelSim Tcl time commands make simulator-time-based values available for use within other Tcl procedures.

Time values may optionally contain a units specifier where the intervening space is also optional. If the space is present, the value must be quoted (e.g. 10ns, "10 ns"). Time values without units are taken to be in the UserTimeScale. Return values are always in the current Time Scale Units. All time values are converted to a 64-bit integer value in the current Time Scale. This means that values smaller than the current Time Scale will be truncated to 0.

Conversions

Command	Description
intToTime <intHi32> <intLo32>	converts two 32-bit pieces (high and low order) into a 64-bit quantity (Time in ModelSim is a 64-bit integer)
RealToTime <real>	converts a <real> number to a 64-bit integer in the current Time Scale
scaleTime <time> <scaleFactor>	returns the value of <time> multiplied by the <scaleFactor> integer

Relations

Command	Description
eqTime <time> <time>	evaluates for equal
neqTime <time> <time>	evaluates for not equal
gtTime <time> <time>	evaluates for greater than
gteTime <time> <time>	evaluates for greater than or equal
ltTime <time> <time>	evaluates for less than
lteTime <time> <time>	evaluates for less than or equal

All relation operations return 1 or 0 for true or false respectively and are suitable return values for TCL conditional expressions. For example,

```
if {[eqTime $Now 1750ns]} {
    ...
}
```

Arithmetic

Command	Description
addTime <time> <time>	add time
divTime <time> <time>	64-bit integer divide
mulTime <time> <time>	64-bit integer multiply
subTime <time> <time>	subtract time

Tcl examples

Example 1

The following Tcl/ModelSim example for UNIX shows how you can access system information and transfer it into VHDL variables or signals and Verilog nets or registers. When a particular HDL source breakpoint occurs, a Tcl function is called that gets the date and time and deposits it into a VHDL signal of type STRING. If a particular environment variable (DO_ECHO) is set, the function also echoes the new date and time to the transcript file by examining the VHDL variable.

Note: In a Windows environment, the Tcl **exec** command shown below will execute compiled files only, not system commands.

(in VHDL source):

```
signal datime : string(1 to 28) := "                            ";# 28 spaces
```

(on VSIM command line or in macro):

```
proc set_date {} {
    global env
    set do_the_echo [set env(DO_ECHO)]
    set s [exec date]
    force -deposit datime $s
    if {do_the_echo} {
        echo "New time is [examine -value datime]"
    }
}
bp src/waveadd.vhd 133 {set_date; continue}
               --sets the breakpoint to call set_date
```

This is an example of using the Tcl **while** loop to copy a list from variable a to variable b, reversing the order of the elements along the way:

```
set b ""
set i [expr[llength $a]-1]
while {$i >= 0} {
        lappend b [lindex $a $i]
        incr i -1
}
```

This example uses the Tcl **for** command to copy a list from variable a to variable b, reversing the order of the elements along the way:

```
set b ""
for {set i [expr [llength $a] -1]} {$i >= 0} {incr i -1} {
    lappend b [lindex $a $i]
}
```

This example uses the Tcl **foreach** command to copy a list from variable a to variable b, reversing the order of the elements along the way (the **foreach** command iterates over all of the elements of a list):

```
set b ""
foreach i $a {
    set b [linsert $b 0 $i]
}
```

This example shows a list reversal as above, this time aborting on a particular element using the Tcl **break** command:

```
set b ""
foreach i $a {
    if {$i = "ZZZ"} break
    set b [linsert $b 0 $i]
}
```

This example is a list reversal that skips a particular element by using the Tcl **continue** command:

```
set b ""
foreach i $a {
    if {$i = "ZZZ"} continue
    set b [linsert $b 0 $i]
}
```

The last example is of the Tcl **switch** command:

```
switch $x {
    a {incr t1}
    b {incr t2}
    c {incr t3}
}
```

Example 2

This next example shows a complete Tcl script that restores multiple Wave windows to their state in a previous simulation, including signals listed, geometry, and screen position. It also adds buttons to the Main window toolbar to ease management of the wave files. This example works in ModelSim SE only.

```
##  This file contains procedures to manage multiple wave files.
##  Source this file from the command line or as a startup script.
##  source <path>/wave_mgr.tcl

##  add_wave_buttons
##      Add wave management buttons to the main toolbar (new, save and load)

##  new_wave
##      Dialog box creates a new wave window with the user provided name

##  named_wave <name>
##      Creates a new wave window with the specified title

##  save_wave <file-root>
##      Saves name, window location and contents for all open windows

##  wave windows
##      Creates <file-root><n>.do file for each window where <n> is 1
##      to the number of windows. Default file-root is "wave". Also
##       creates windowSet.do file that contains title and geometry info.

##  load_wave <file-root>
##      Opens and loads wave windows for all files matching <file-root><n>.do
##      where <n> are the numbers from 1-9. Default <file-root> is "wave".
##       Also runs windowSet.do file if it exists.
```

```
## Add wave management buttons to the main toolbar

proc add_wave_buttons {} {
_add_menu main controls right SystemMenu SystemWindowFrame {Load Waves} \
load_wave
_add_menu main controls right SystemMenu SystemWindowFrame {Save Waves} \
save_wave
_add_menu main controls right SystemMenu SystemWindowFrame {New Wave} \
new_wave
}

## Simple Dialog requests name of new wave window.  Defaults to Wave<n>

proc new_wave {} {
    global dialog_prompt vsimPriv
    set defaultName "Wave[llength $vsimPriv(WaveWindows)]"
    set dialog_prompt(result) $defaultName
    set windowName [GetValue . "Create Named Wave Window:" ]
    ## Debug
    puts "Window name:  $windowName\n";
    if {$windowName == "{}"} {
      set windowName ""
    }
    if {$windowName != ""} {
      named_wave $windowName
    } else {
      named_wave $defaultName
    }
}

## Creates a new wave window with the provided name (defaults to "Wave")

proc named_wave {{name "Wave"}} {
    global vsimPriv
    view -new wave
    set newWave [lindex $vsimPriv(WaveWindows) [expr [llength \
    $vsimPriv(WaveWindows)] - 1]]
    wm title $newWave $name
}

## Writes out format of all wave windows, stores geometry and title info in
## windowSet.do file.  Removes any extra files with the same fileroot.
## Default file name is wave<n> starting from 1.

proc save_wave {{fileroot "wave"}} {
    global vsimPriv
    set n 1
    set fileId [open windowSet_$fileroot.do w 755]
    foreach w $vsimPriv(WaveWindows) {
        echo "Saving: [wm title $w]"
        set filename $fileroot$n.do
        write format wave -window $w $filename
        puts $fileId "wm title $w \"[wm title $w]\""
        puts $fileId "wm geometry $w [wm geometry $w]"
        puts $fileId "mtiGrid_colconfig $w.grid name -width \
        [mtiGrid_colcget $w.grid name -width]"
        puts $fileId "mtiGrid_colconfig $w.grid value -width \
        [mtiGrid_colcget $w.grid value -width]"
        flush $fileId
        incr n
    }
```

```
    if {![catch {glob $fileroot\[$n-9\].do}]} {
        foreach f [lsort [glob $fileroot\[$n-9\].do]] {
            echo "Removing:  $f"
            exec rm $f
        }
    }
}

## Provide file root argument and load_wave restores all saved windows.
## Default file root is "wave".

proc load_wave {{fileroot "wave"}} {
    global vsimPriv
    foreach f [lsort [glob $fileroot\[1-9\].do]] {
        echo "Loading: $f"
        view -new wave
        do $f
    }
    if {[file exists windowSet_$fileroot.do]} {
        do windowSet_$fileroot.do
    }
}
```

Macros (DO files)

ModelSim macros (also called DO files) are simply scripts that contain ModelSim and, optionally, Tcl commands. You invoke these scripts with the **Tools > Execute Macro** (Main window) menu selection or the **do** command (CR-68).

Creating DO files

You can create DO files, like any other Tcl script, by typing the required commands in any editor and saving the file. Alternatively, you can save the Main window transcript as a DO file (see "Saving the Main window transcript file" (UM-139)).

The following is a simple DO file that was saved from the Main window transcript. It is used in the dataset exercise in the ModelSim Tutorial. This DO file adds several signals to the Wave window, provides stimulus to those signals, and then advances the simulation.

```
add wave ld
add wave rst
add wave clk
add wave d
add wave q
force -freeze clk 0 0, 1 {50 ns} -r 100
force rst 1
force rst 0 10
force ld 0
force d 1010
run 1700
force ld 1
run 100
force ld 0
run 400
force rst 1
run 200
force rst 0 10
run 1500
```

Using Parameters with DO files

You can increase the flexibility of DO files by using parameters. Parameters specify values that are passed to the corresponding parameters $1 through $9 in the macro file. For example say the macro "*testfile*" contains the line **bp** $1 $2. The command below would place a breakpoint in the source file named *design.vhd* at line 127:

```
do  testfile  design.vhd  127
```

There is no limit on the number of parameters that can be passed to macros, but only nine values are visible at one time. You can use the **shift** command (CR-118) to see the other parameters.

Making macro parameters optional

If you want to make macro parameters optional (i.e., be able to specify fewer parameter values with the do command than the number of parameters referenced in the macro), you must use the argc (UM-353) simulator state variable. The **argc** simulator state variable returns the number of parameters passed. The examples below show several ways of using **argc**.

Example 1

This macro specifies the files to compile and handles 0-2 compiler arguments as parameters. If you supply more arguments, ModelSim generates a message.

```
switch $argc {
  0 {vcom file1.vhd file2.vhd file3.vhd }
  1 {vcom $1 file1.vhd file2.vhd file3.vhd }
  2 {vcom $1 $2 file1.vhd file2.vhd file3.vhd }
  default {echo Too many arguments. The macro accepts 0-2 args.  }
}
```

Example 2

This macro specifies the compiler arguments and lets you compile any number of files.

```
variable Files ""
set nbrArgs $argc
for {set x 1} {$x <= $nbrArgs} {incr x} {
  set Files [concat $Files $1]
  shift
}
eval vcom -93 -explicit -noaccel $Files
```

Example 3

This macro is an enhanced version of the one shown in example 2. The additional code determines whether the files are VHDL or Verilog and uses the appropriate compiler and arguments depending on the file type. Note that the macro assumes your VHDL files have a .vhd file extension.

```
variable vhdFiles ""
variable vFiles ""
set nbrArgs $argc
set vhdFilesExist 0
set vFilesExist   0
for {set x 1} {$x <= $nbrArgs} {incr x} {
  if {[string match *.vhd $1]} {
    set vhdFiles [concat $vhdFiles $1]
    set vhdFilesExist 1
  } else {
    set vFiles [concat $vFiles $1]
    set vFilesExist 1
  }
  shift
}
if {$vhdFilesExist == 1} {
  eval vcom -93 -explicit -noaccel $vhdFiles
}
if {$vFilesExist == 1} {
  eval vlog -fast -forcecode $vFiles
}
```

Useful commands for handling breakpoints and errors

If you are executing a macro when your simulation hits a breakpoint or causes a run-time error, ModelSim interrupts the macro and returns control to the command line. The following commands may be useful for handling such events. (Any other legal command may be executed as well.)

command	result
run (CR-114) **-continue**	continue as if the breakpoint had not been executed, completes the **run** (CR-114) that was interrupted
onbreak (CR-98)	specify a command to run when you hit a breakpoint within a macro
onElabError (CR-99)	specify a command to run when an error is encountered during elaboration
onerror (CR-100)	specify a command to run when an error is encountered within a macro
status (CR-121)	get a traceback of nested macro calls when a macro is interrupted
abort (CR-30)	terminate a macro once the macro has been interrupted or paused
pause (CR-101)	cause the macro to be interrupted; the macro can be resumed by entering a **resume** command (CR-113) via the command line

Note: You can also set the OnErrorDefaultAction Tcl variable (see "Preference variables located in Tcl files" (UM-352)) in the *pref.tcl* file to dictate what action ModelSim takes when an error occurs.

Error action in DO files

If a command in a macro returns an error, ModelSim does the following:

1. If an **onerror** (CR-100) command has been set in the macro script, ModelSim executes that command.
2. If no **onerror** command has been specified in the script, ModelSim checks the **OnErrorDefaultAction** Tcl variable. If the variable is defined, it's action will be invoked.
3. If neither 1 or 2 is true, the macro aborts.

Using the Tcl source command with DO files

Either the **do** command or Tcl **source** command can execute a DO file, but they behave differently.

With the **source** command, the DO file is executed exactly as if the commands in it were typed in by hand at the prompt. Each time a breakpoint is hit the Source window is updated to show the breakpoint. This behavior could be inconvenient with a large DO file containing many breakpoints.

When a **do** command is interrupted by an error or breakpoint, it does not update any windows, and keeps the DO file "locked". This keeps the Source window from flashing, scrolling, and moving the arrow when a complex DO file is executed. Typically an **onbreak resume** command is used to keep the macro running as it hits breakpoints. Add an **onbreak abort** command to the DO file if you want to exit the macro and update the Source window.

A - ModelSim variables

Appendix contents

Variable settings report UM-336

Personal preferences UM-336

Returning to the original ModelSim defaults UM-337

Environment variables UM-337

Preference variables located in INI files UM-341
- [Library] library path variables UM-341
- [vcom] VHDL compiler control variables UM-342
- [vlog] Verilog compiler control variables UM-343
- [vsim] simulator control variables UM-344
- Commonly used INI variables UM-349
- Commonly used INI variables UM-349

Preference variables located in Tcl files UM-352

Variable precedence UM-353

Simulator state variables UM-353
- Referencing simulator state variables UM-354
- Special considerations for the now variable UM-354

This appendix documents the following types of ModelSim variables:

- **environment variables**
  Variables referenced and set according to operating system conventions. Environment variables prepare the ModelSim environment prior to simulation.
- **ModelSim preference variables**
  Variables used to control compiler or simulator functions and modify the appearance of the ModelSim GUI.
- **simulator state variable**s
  Variables that provide feedback on the state of the current simulation.

Variable settings report

The **report** command (CR-109) returns a list of current settings for either the simulator state, or simulator control variables. Use the following commands at either the ModelSim or VSIM prompt:

```
report simulator state
report simulator control
```

The simulator control variables reported by the **report simulator control** command can be set interactively using the Tcl **set** command (UM-321).

Personal preferences

There are several preferences stored by ModelSim on a personal basis, independent of *modelsim.ini* or *modelsim.tcl* files. These preferences are stored in the Windows Registry under HKEY_CURRENT_USER\Software\Model Technology Incorporated\ModelSim.

- **cwd**
  History of the last five working directories (pwd). This history appears in the Main window File menu.
- **datasets**
  History of previously opened datasets. Used to populate the **Dataset Pathname** list box in the **Open Dataset** dialog.
- **mti_ask_LBViewTypes**, **mti_ask_LBViewPath**, **mti_ask_LBViewLoadable**
  Settings for the **Customize Library View** dialog. Determine the view of the Library tab in the Main window workspace.
- **mti_pane_cnt**, **mti_pane_size**, **pane_#**, **pane_percent**
  Determine layout of various panes in the Main window.
- **open_workspace**
  Setting for whether or not to display the Main window workspace.
- **pinit**
  Project Initialization state (one of: Welcome | OpenLast | NoWelcome). This determines whether the Welcome To ModelSim dialog box appears when you invoke the tool.
- **project_history**
  Project History
- **printersetup**
  All setup parameters related to Printing (i.e., current printer, etc.)
- **transcriptpercent**
  The size of the Main window transcript pane. Expressed as a percentage of the width of the Main window.

The HKEY_CURRENT_USER key is unique for each user Login on Windows NT.

Returning to the original ModelSim defaults

If you would like to return ModelSim's interface to its original state, simply rename or delete the existing *modelsim.tcl* and *modelsim.ini* files. ModelSim will use *pref.tcl* for GUI preferences and make a copy of *<install_dir>/modeltech/modelsim.ini* to use the next time ModelSim is invoked without an existing project (if you start a new project the new MPF file will use the settings in the new *modelsim.ini* file).

Environment variables

Before compiling or simulating, several environment variables may be set to provide the functions described in the table below. The variables are in the *autoexec.bat* file on Windows 98/Me machines, and set through the System control panel on NT/2000 machines. The LM_LICENSE_FILE variable is required; all others are optional.

ModelSim Environment Variables

Variable	Description
DOPATH	used by ModelSim to search for DO files (macros); consists of a colon-separated (semi-colon for Windows) list of paths to directories; this environment variable can be overridden by the DOPATH Tcl preference variable The DOPATH environment variable isn't accessible when you invoke **vsim** from a Unix shell or from a Windows command prompt. It is accessible once ModelSim or **vsim** is invoked. If you need to invoke from a shell or command line and use the DOPATH environment variable, use the following syntax: `vsim -do "do <dofile_name>" <design_unit>`
EDITOR	specifies the editor to invoke with the **edit** command (CR-72)
HOME	used by ModelSim to look for an optional graphical preference file and optional location map file; see: "Preference variables located in INI files" (UM-341) and "Using location mapping" (UM-387)
LM_LICENSE_FILE	used by the ModelSim license file manager to find the location of the license file; may be a colon-separated (semi-colon for Windows) set of paths, including paths to other vendor license files; REQUIRED
MODEL_TECH	set by all ModelSim tools to the directory in which the binary executable resides; **DO NOT SET THIS VARIABLE!**
MODEL_TECH_TCL	used by ModelSim to find Tcl libraries for Tcl/Tk 8.3 and vsim; may also be used to specify a startup DO file; defaults to */modeltech/../tcl*; may be set to an alternate path
MGC_LOCATION_MAP	used by ModelSim tools to find source files based on easily reallocated "soft" paths; optional; see: "Using location mapping" (UM-387); also see the Tcl variables: **SourceDir** and **SourceMap**

Variable	Description
MODELSIM	used by all ModelSim tools to find the *modelsim.ini* file; consists of a path including the file name. An alternative use of this variable is to set it to the path of a project file (<*Project_Root_Dir*>/<*Project_Name*>.*mpf*). This allows you to use project settings with command line tools. However, if you do this, the .mpf file will replace *modelsim.ini* as the initialization file for all ModelSim tools.
MODELSIM_TCL	used by ModelSim to look for an optional graphical preference file; can be a semi-colon separated (Windows) list of file paths
MTI_COSIM_TRACE	creates an *mti_trace_cosim* file containing debugging information about FLI/PLI/VPI function calls; set to any value before invoking the simulator.
MTI_TF_LIMIT	limits the size of the VSOUT temp file (generated by the ModelSim kernel); the value of the variable is the size of k-bytes; TMPDIR (below) controls the location of this file, STDOUT controls the name; default = 10, 0 = no limit; does *not* control the size of the transcript file
MTI_USELIB_DIR	specifies the directory into which object libraries are compiled when using the **-compile_uselibs** argument to the **vlog** command (CR-181)
NOMMAP	if set to 1, disables memory mapping in ModelSim; this should be used only when running on Linux 7.1; it will decrease the speed with which ModelSim reads files
PLIOBJS	used by ModelSim to search for PLI object files for loading; consists of a space-separated list of file or path names
STDOUT	the VSOUT temp file (generated by the simulator kernel) is deleted when the simulator exits; the file is not deleted if you specify a filename for VSOUT with STDOUT; specifying a name and location (use TMPDIR) for the VSOUT file will also help you locate and delete the file in event of a crash (an unnamed VSOUT file is not deleted after a crash either)
TMP	specifies the path to a tempnam() generated file (VSOUT) containing all stdout from the simulation kernel

Creating environment variables in Windows

In addition to the predefined variables shown above, you can define your own environment variables. This example shows a user-defined library path variable that can be referenced by the **vmap** command to add library mapping to the *modelsim.ini* file.

Using Windows 98/Me

Open and edit the *autoexec.bat* file by adding this line:

```
set MY_PATH=\temp\work
```

Restart Windows to initialize the new variable.

Using Windows NT/2000/XP

Right-click the **My Computer** icon and select **Properties**, then select the **Environment** tab (in Windows 2000/XP select the Advanced tab and then Environment Variables). Add the new variable with this data—Variable:*MY_PATH* and Value:*\temp\work*.

Click **Set** and **Apply** to initialize the variable.

Library mapping with environment variables

Once the **MY_PATH** variable is set, you can use it with the **vmap** command (CR-188) to add library mappings to the current *modelsim.ini* file.

If you're using the **vmap** command from DOS prompt type:

```
vmap MY_VITAL %MY_PATH%
```

If you're using **vmap** from the ModelSim/VSIM prompt type:

```
vmap MY_VITAL \$MY_PATH
```

If you used DOS **vmap**, this line will be added to the *modelsim.ini*:

```
MY_VITAL = c:\temp\work
```

If **vmap** is used from the ModelSim/VSIM prompt, the *modelsim.ini* file will be modified with this line:

```
MY_VITAL = $MY_PATH
```

You can easily add additional hierarchy to the path. For example,

```
vmap MORE_VITAL %MY_PATH%\more_path\and_more_path
vmap MORE_VITAL \$MY_PATH\more_path\and_more_path
```

The "$" character in the examples above is Tcl syntax that precedes a variable. The "\" character is an escape character that keeps the variable from being evaluated during the execution of **vmap**.

Referencing environment variables within ModelSim

There are two ways to reference environment variables within ModelSim. Environment variables are allowed in a **FILE** variable being opened in VHDL. For example,

```
use std.textio.all;
entity test is end;
architecture only of test is
begin
    process
        FILE in_file : text is in "$ENV_VAR_NAME";
    begin
        wait;
    end process;
end;
```

Environment variables may also be referenced from the ModelSim command line or in macros using the Tcl **env** array mechanism:

```
echo "$env(ENV_VAR_NAME)"
```

Note: Environment variable expansion *does not* occur in files that are referenced via the **-f** argument to **vcom**, **vlog**, or **vsim**.

Removing temp files (VSOUT)

The *VSOUT* temp file is the communication mechanism between the simulator kernel and the ModelSim GUI. In normal circumstances the file is deleted when the simulator exits. If ModelSim crashes, however, the temp file must be deleted manually. Specifying the location of the temp file with **TMPDIR** (above) will help you locate and remove the file.

Preference variables located in INI files

ModelSim initialization (INI) files contain control variables that specify reference library paths and compiler and simulator settings. The default initialization file is *modelsim.ini* and is located in your install directory.

To set these variables, edit the initialization file directly with any text editor. The syntax for variables in the file is:

```
<variable> = <value>
```

Comments within the file are preceded with a semicolon (;).

The following tables list the variables by section, and in order of their appearance within the INI file:

INI file sections
[Library] library path variables (UM-341)
[vcom] VHDL compiler control variables (UM-342)
[vlog] Verilog compiler control variables (UM-343)
[vsim] simulator control variables (UM-344)

[Library] library path variables

Variable name	Value range	Purpose
ieee	any valid path; may include environment variables	sets the path to the library containing IEEE and Synopsys arithmetic packages; the default is $MODEL_TECH/../ieee
modelsim_lib	any valid path; may include environment variables	sets the path to the library containing Model Technology VHDL utilities such as Signal Spy; the default is $MODEL_TECH/../modelsim_lib
std	any valid path; may include environment variables	sets the path to the VHDL STD library; the default is $MODEL_TECH/../std
std_developerskit	any valid path; may include environment variables	sets the path to the libraries for MGC standard developer's kit; the default is $MODEL_TECH/../std_developerskit
synopsys	any valid path; may include environment variables	sets the path to the accelerated arithmetic packages; the default is $MODEL_TECH/../ synopsys
verilog	any valid path; may include environment variables	sets the path to the library containing VHDL/ Verilog type mappings; the default is $MODEL_TECH/../verilog
vital2000	any valid path; may include environment variables	sets the path to the VITAL 2000 library; the default is $MODEL_TECH/../vital2000

Variable name	Value range	Purpose
others	any valid path; may include environment variables	points to another *modelsim.ini* file whose library path variables will also be read; the path name must include "modelsim.ini"; only one others variable can be specified in any *modelsim.ini* file.

[vcom] VHDL compiler control variables

Variable name	Value range	Purpose	Default
CheckSynthesis	0, 1	if 1, turns on limited synthesis rule compliance checking; checks only signals used (read) by a process; also, understands only combinational logic, not clocked logic	off (0)
Explicit	0, 1	if 1, turns on resolving of ambiguous function overloading in favor of the "explicit" function declaration (not the one automatically created by the compiler for each type declaration)	on (1)
IgnoreVitalErrors	0, 1	if 1, ignores VITAL compliance checking errors	off (0)
NoCaseStaticError	0, 1	if 1, changes case statement static errors to warnings	off (0)
NoDebug	0, 1	if 1, turns off inclusion of debugging info within design units	off (0)
NoIndexCheck	0, 1	if 1, run time index checks are disabled	off (0)
NoOthersStaticError	0, 1	if 1, disables errors caused by aggregates that are not locally static	off (0)
NoRangeCheck	0, 1	if 1, disables run time range checking	off (0)
NoVital	0, 1	if 1, turns off acceleration of the VITAL packages	off (0)
NoVitalCheck	0, 1	if 1, turns off VITAL compliance checking	off (0)
Optimize_1164	0, 1	if 0, turns off optimization for IEEE std_logic_1164 package	on (1)
PedanticErrors	0, 1	if 1, overrides NoCaseStaticError and NoOthersStaticError	off(0)
Quiet	0, 1	if 1, turns off "loading..." messages	off (0)
RequireConfigForAllDefaultBinding	0, 1	if 1, instructs the compiler not to generate a default binding during compilation	off (0)
Show_source	0, 1	if 1, shows source line containing error	off (0)
Show_VitalChecksWarnings	0, 1	if 0, turns off VITAL compliance-check warnings	on (1)

Variable name	Value range	Purpose	Default
Show_Warning1	0, 1	if 0, turns off unbound-component warnings	on (1)
Show_Warning2	0, 1	if 0, turns off process-without-a-wait-statement warnings	on (1)
Show_Warning3	0, 1	if 0, turns off null-range warnings	on (1)
Show_Warning4	0, 1	if 0, turns off no-space-in-time-literal warnings	on (1)
Show_Warning5	0, 1	if 0, turns off multiple-drivers-on-unresolved-signal warnings	on (1)
VHDL93	0, 1	if 1, turns on VHDL-1993	off (0)

[vlog] Verilog compiler control variables

Variable name	Value range	Purpose	Default
Hazard	0, 1	if 1, turns on Verilog hazard checking (order-dependent accessing of global variables)	off (0)
Incremental	0, 1	if 1, turns on incremental compilation of modules	off (0)
NoDebug	0, 1	if 1, turns off inclusion of debugging info within design units	off (0)
Quiet	0, 1	if 1, turns off "loading..." messages	off (0)
Show_Lint	0, 1	if 1, turns on lint-style checking	off (0)
Show_source	0, 1	if 1, shows source line containing error	off (0)
vlog95compat	0, 1	if 1, disables Verilog 2001 support and makes compiler compatible with IEEE Std 1364-1995	off (0)

[vsim] simulator control variables

Variable name	Value range	Purpose	Default
AssertFile	any valid filename	alternative file for storing assertion messages	transcript
AssertionFormat	see next column	defines format of assertion messages; fields include: %S - severity level %R - report message %T - time of assertion %D - delta %I - instance or region pathname (if available) %i - instance pathname with process %O - process name %K - kind of item path points to; returns Instance, Signal, Process, or Unknown %P - instance or region path without leaf process %F - file %L - line number of assertion, or if from subprogram, line from which call is made %% - print '%' character	"** %S: %R\n Time: %T Iteration: %D%I\n"
AssertionFormatBreak	see AssertionFormat above	defines format of messages for assertions that trigger a breakpoint; see AssertionFormat for options;	"** %S: %R\n Time: %T Iteration: %D %K: %i File: %F\n"
AssertionFormatNote	see AssertionFormat above	defines format of messages for Note assertions; see AssertionFormat for options; if undefined, AssertionFormat is used unless assertion causes a breakpoint in which case AssertionFormatBreak is used	"** %S: %R\n Time: %T Iteration: %D%I\n"
AssertionFormatWarning	see AssertionFormat above	defines format of messages for Warning assertions; see AssertionFormat for options; if undefined, AssertionFormat is used unless assertion causes a breakpoint in which case AssertionFormatBreak is used	"** %S: %R\n Time: %T Iteration: %D%I\n"
AssertionFormatError	see AssertionFormat above	defines format of messages for Error assertions; see AssertionFormat for options; if undefined, AssertionFormat is used unless assertion causes a breakpoint in which case AssertionFormatBreak is used	"** %S: %R\n Time: %T Iteration: %D %K: %i File: %F\n"

Variable name	Value range	Purpose	Default
AssertionFormatFail	see AssertionFormat above	defines format of messages for Fail assertions; see AssertionFormat for options; if undefined, AssertionFormat is used unless assertion causes a breakpoint in which case AssertionFormatBreak is used	"** %S: %R\n Time: %T Iteration: %D %K: %i File: %F\n"
AssertionFormatFatal	see AssertionFormat above	defines format of messages for Fatal assertions; see AssertionFormat for options; if undefined, AssertionFormat is used unless assertion causes a breakpoint in which case AssertionFormatBreak is used	"** %S: %R\n Time: %T Iteration: %D %K: %i File: %F\n"
BreakOnAssertion	0-4	defines severity of assertion that causes a simulation break (0 = note, 1 = warning, 2 = error, 3 = failure, 4 = fatal); this variable can be set interactively with the Tcl **set** command (UM-321)	3
CheckpointCompressMode	0, 1	if 1, checkpoint files are written in compressed format; this variable can be set interactively with the Tcl **set** command (UM-321)	on (1)
CommandHistory	any valid filename	sets the name of a file in which to store the Main window command history	commented out (;)
ConcurrentFileLimit	any positive integer	controls the number of VHDL files open concurrently; this number should be less than the current limit setting for max file descriptors; 0 = unlimited	40
DatasetSeparator	any character except those with special meaning (i.e., \, {, }, etc.)	the dataset separator for fully-rooted contexts, for example sim:/top; must not be the same character as PathSeparator	:
DefaultForceKind	freeze, drive, or deposit	defines the kind of force used when not otherwise specified; this variable can be set interactively with the Tcl **set** command (UM-321)	drive for resolved signals; freeze for unresolved signals
DefaultRadix	symbolic, binary, octal, decimal, unsigned, hexadecimal, ascii	a numeric radix may be specified as a name or number (i.e., binary can be specified as binary or 2; octal as octal or 8; etc.); this variable can be set interactively with the Tcl **set** command (UM-321)	symbolic

Variable name	Value range	Purpose	Default
DefaultRestartOptions	one or more of: -force, -nobreakpoint, -nolist, -nolog, -nowave	sets default behavior for the restart command	commented out (;)
DelayFileOpen	0, 1	if 1, open VHDL87 files on first read or write, else open files when elaborated; this variable can be set interactively with the Tcl **set** command (UM-321)	off (0)
GenerateFormat	Any non-quoted string containing at a minimum a %s followed by a %d	controls the format of a generate statement label (don't quote it)	%s__%d
IgnoreError	0,1	if 1, ignore assertion errors; this variable can be set interactively with the Tcl **set** command (UM-321)	off (0)
IgnoreFailure	0,1	if 1, ignore assertion failures; this variable can be set interactively with the Tcl **set** command (UM-321)	off (0)
IgnoreNote	0,1	if 1, ignore assertion notes; this variable can be set interactively with the Tcl **set** command (UM-321)	off (0)
IgnoreWarning	0,1	if 1, ignore assertion warnings; this variable can be set interactively with the Tcl **set** command (UM-321)	off (0)
IterationLimit	positive integer	limit on simulation kernel iterations allowed without advancing time; this variable can be set interactively with the Tcl **set** command (UM-321)	5000

Variable name	Value range	Purpose	Default
License	any single <license_option>	if set, controls ModelSim license file search; license options include: nomgc - excludes MGC licenses nomti - excludes MTI licenses noqueue - do not wait in license queue if no licenses are available plus - only use PLUS license vlog - only use VLOG license vhdl - only use VHDL license viewsim - accepts a simulation license rather than being queued for a viewer license see also the **vsim** command (CR-189) <license_option>	search all licenses
NumericStdNoWarnings	0, 1	if 1, warnings generated within the accelerated numeric_std and numeric_bit packages are suppressed; this variable can be set interactively with the Tcl **set** command (UM-321).	off (0)
PathSeparator	any character except those with special meaning (i.e., \, {, }, etc.)	used for hierarchical path names; must not be the same character as DatasetSeparator; this variable can be set interactively with the Tcl **set** command (UM-321)	/
Resolution	fs, ps, ns, us, ms, or sec with optional prefix of 1, 10, or 100	simulator resolution; no space between value and units (i.e., 10fs, not 10 fs); overridden by the -t argument to **vsim** (CR-189); if your delays get truncated, set the resolution smaller; this value must be less than or equal to the UserTimeUnit (described below)	ps
RunLength	positive integer	default simulation length in units specified by the UserTimeUnit variable; this variable can be set interactively with the Tcl **set** command (UM-321).	100
Startup	= do <DO filename>; any valid macro (do) file	specifies the ModelSim startup macro; see the **do** command (CR-68)	commented out (;)
StdArithNoWarnings	0, 1	if 1, warnings generated within the accelerated Synopsys std_arith packages are suppressed; this variable can be set interactively with the Tcl **set** command (UM-321)	off (0)

Variable name	Value range	Purpose	Default
TranscriptFile	any valid filename	file for saving command transcript; environment variables may be included in the path name	transcript
UnbufferedOutput	0, 1	controls VHDL and Verilog files open for write; 0 = Buffered, 1 = Unbuffered	0
UserTimeUnit	fs, ps, ns, us, ms, sec, or default	specifies scaling for the Wave window and the default time units to use for commands such as **force** (CR-82) and **run** (CR-114); should generally be set to default, in which case it takes the value of the Resolution variable; this variable can be set interactively with the Tcl **set** command (UM-321)	default
Veriuser	one or more valid shared object names	list of dynamically loadable objects for Verilog PLI/VPI applications; see "Verilog PLI/VPI" (UM-97)	commented out (;)
WaveSignalNameWidth	0, positive integer	controls the number of visible hierarchical regions of a signal name shown in the Wave window (UM-206); the default value of zero displays the full name, a setting of one or above displays the corresponding level(s) of hierarchy	0
WLFCompress	0, 1	turns WLF file compression on (1) or off (0)	1
WLFDeleteOnQuit	0, 1	specifies whether a WLF file should be deleted when the simulation ends; if set to 0, the file is not deleted; if set to 1, the file is deleted	0
WLFSaveAllRegions	0, 1	specifies whether to save all design hierarchy in the WLF file (1) or only regions containing logged signals (0)	0
WLFSizeLimit	0 - positive integer of MB	WLF file size limit; limits WLF file by size (as closely as possible) to the specified number of megabytes; if both size and time limits are specified the most restrictive is used; setting to 0 results in no limit	0
WLFTimeLimit	0 - positive integer of MB	WLF file time limit; limits WLF file by time (as closely as possible) to the specified amount of time. If both time and size limits are specified the most restrictive is used; setting to 0 results in no limit	0

Commonly used INI variables

Several of the more commonly used *modelsim.ini* variables are further explained below.

Environment variables

You can use environment variables in your initialization files. Use a dollar sign ($) before the environment variable name. For example:

```
[Library]
work = $HOME/work_lib
test_lib = ./$TESTNUM/work
...
[vsim]
IgnoreNote = $IGNORE_ASSERTS
IgnoreWarning = $IGNORE_ASSERTS
IgnoreError = 0
IgnoreFailure = 0
```

There is one environment variable, MODEL_TECH, that you cannot — and should not — set. MODEL_TECH is a special variable set by Model Technology software. Its value is the name of the directory from which the VCOM or VLOG compilers or VSIM simulator was invoked. MODEL_TECH is used by the other Model Technology tools to find the libraries.

Hierarchical library mapping

By adding an "others" clause to your *modelsim.ini* file, you can have a hierarchy of library mappings. If the ModelSim tools don’t find a mapping in the *modelsim.ini* file, then they will search only the library section of the initialization file specified by the "others" clause. For example:

```
[Library]
asic_lib = /cae/asic_lib
work = my_work
others = /install_dir/modeltech/modelsim.ini
```

Since the file referred to by the "others" clause may itself contain an "others" clause, you can use this feature to chain a set of hierarchical INI files for library mappings.

Creating a transcript file

A feature in the system initialization file allows you to keep a record of everything that occurs in the transcript: error messages, assertions, commands, command outputs, etc. To do this, set the value for the TranscriptFile line in the *modelsim.ini* file to the name of the file in which you would like to record the ModelSim history.

```
; Save the command window contents to this file
TranscriptFile = trnscrpt
```

Using a startup file

The system initialization file allows you to specify a command or a *do* file that is to be executed after the design is loaded. For example:

```
; VSIM Startup command
Startup = do mystartup.do
```

The line shown above instructs ModelSim to execute the commands in the macro file named *mystartup.do*.

```
; VSIM Startup command
Startup = run -all
```

The line shown above instructs VSIM to run until there are no events scheduled.

See the **do** command (CR-68) for additional information on creating do files.

Turning off assertion messages

You can turn off assertion messages from your VHDL code by setting a switch in the *modelsim.ini* file. This option was added because some utility packages print a huge number of warnings.

```
[vsim]
IgnoreNote = 1
IgnoreWarning = 1
IgnoreError = 1
IgnoreFailure = 1
```

Turning off warnings from arithmetic packages

You can disable warnings from the Synopsys and numeric standard packages by adding the following lines to the [vsim] section of the *modelsim.ini* file.

```
[vsim]
NumericStdNoWarnings = 1
StdArithNoWarnings = 1
```

These variables can also be set interactively using the Tcl **set** command (UM-321). This capability provides an answer to a common question about disabling warnings at time 0. You might enter commands like the following in a DO file or at the ModelSim prompt:

```
set NumericStdNoWarnings 1
run 0
set NumericStdNoWarnings 0
run -all
```

Alternatively, you could use the **when** command (CR-205) to accomplish the same thing:

```
when {$now = @1ns } {set NumericStdNoWarnings 1}
run -all
```

Note that the time unit (ns in this case) would vary depending on your simulation resolution.

Force command defaults

The **force** command has **-freeze**, **-drive**, and **-deposit** options. When none of these is specified, then **-freeze** is assumed for unresolved signals and **-drive** is assumed for resolved signals. This is designed to provide compatibility with force files. But if you prefer **-freeze** as the default for both resolved and unresolved signals, you can change the defaults in the *modelsim.ini* file.

```
[vsim]
; Default Force Kind
; The choices are freeze, drive, or deposit
DefaultForceKind = freeze
```

Restart command defaults

The **restart** command has **-force**, **-nobreakpoint**, **-nolist**, **-nolog**, and **-nowave** options. You can set any of these as defaults by entering the following line in the *modelsim.ini* file:

```
DefaultRestartOptions = <options>
```

where `<options>` can be one or more of -force, -nobreakpoint, -nolist, -nolog, and -nowave.

Example: `DefaultRestartOptions = -nolog -force`

Note: You can also set these defaults in the *modelsim.tcl* file. The Tcl file settings will override the .ini file settings.

VHDL93

You can make the VHDL93 standard the default by including the following line in the *INI* file:

```
[vcom]
; Turn on VHDL-1993 as the default. Default is off (VHDL-1987).
VHDL93 = 1
```

Opening VHDL files

You can delay the opening of VHDL files with an entry in the *INI* file if you wish. Normally VHDL files are opened when the file declaration is elaborated. If the **DelayFileOpen** option is enabled, then the file is not opened until the first read or write to that file.

```
[vsim]
DelayFileOpen = 1
```

Preference variables located in Tcl files

ModelSim Tcl preference variables give you control over fonts, colors, prompts, window positions and other simulator window characteristics. Preference files, which contain Tcl commands that set preference variables, are loaded before any windows are created, and so will affect all windows.

When ModelSim is invoked for the first time, default preferences are loaded from the *pref.tcl* file. Customized variable settings may be set from within the ModelSim GUI (**Tools > Edit Preferences** (Main window)), on the ModelSim command line (with the Tcl **set** command (UM-321)), or by directly editing the preference file.

The default file for customized preferences is *modelsim.tcl*. When ModelSim starts it searches for a *modelsim.tcl* file as follows:

- use MODELSIM_TCL (UM-338) environment variable if it exists (if MODELSIM_TCL is a list of files, each file is loaded in the order that it appears in the list); else
- use *./modelsim.tcl*; else
- use $(HOME)/modelsim.tcl if it exists

Important: If your preference file is not named *modelsim.tcl*, or if the file is not located in the directories mentioned above, you must refer to it with the MODELSIM_TCL environment variable.

For complete documentation on each Tcl preference variables, see the following URL:

http://www.model.com/resources/pref_variables/frameset.htm

User-defined variables

Temporary, user-defined variables can be created with the Tcl **set** command (UM-321). Like simulator variables, user-defined variables are preceded by a dollar sign when referenced. To create a variable with the **set** command:

```
set user1 7
```

You can use the variable in a command like:

```
echo "user1 = $user1"
```

More preferences

Additional compiler and simulator preferences may be set in the *modelsim.ini* file; see "Preference variables located in INI files" (UM-341).

Variable precedence

Note that some variables can be set in a .tcl file or a .ini file. A variable set in a .tcl file takes precedence over the same variable set in a .ini file. For example, assume you have the following line in your *modelsim.ini* file:

```
TranscriptFile = transcript
```

And assume you have the following line in your *modelsim.tcl* file:

```
set PrefMain(file) {}
```

In this case the setting in the *modelsim.tcl* file will override that in the *modelsim.ini* file, and a transcript file will not be produced.

Simulator state variables

Unlike other variables that must be explicitly set, simulator state variables return a value relative to the current simulation. Simulator state variables can be useful in commands, especially when used within ModelSim DO files (macros).

Variable	Result
argc	returns the total number of parameters passed to the current macro
architecture	returns the name of the top-level architecture currently being simulated; for a configuration or Verilog module, this variable returns an empty string
configuration	returns the name of the top-level configuration currently being simulated; returns an empty string if no configuration
delta	returns the number of the current simulator iteration
entity	returns the name of the top-level VHDL entity or Verilog module currently being simulated
library	returns the library name for the current region
MacroNestingLevel	returns the current depth of macro call nesting
n	represents a macro parameter, where n can be an integer in the range 1-9
Now	always returns the current simulation time with time units (e.g., 110,000 ns) Note: will return a comma between thousands
now	when time resolution is a unary unit (i.e., 1ns, 1ps, 1fs): returns the current simulation time without time units (e.g., 100000) when time resolution is a multiple of the unary unit (i.e., 10ns, 100ps, 10fs): returns the current simulation time with time units (e.g. 110000 ns) Note: will not return comma between thousands
resolution	returns the current simulation time resolution

Referencing simulator state variables

Variable values may be referenced in simulator commands by preceding the variable name with a dollar sign ($). For example, to use the **now** and **resolution** variables in an **echo** command type:

```
echo "The time is $now $resolution."
```

Depending on the current simulator state, this command could result in:

```
The time is 12390 10ps.
```

If you do not want the dollar sign to denote a simulator variable, precede it with a "\". For example, \$now will not be interpreted as the current simulator time.

Special considerations for the now variable

For the **when** command (CR-205), special processing is performed on comparisons involving the **now** variable. If you specify "when {$now=100}...", the simulator will stop at time 100 regardless of the multiplier applied to the time resolution.

You must use 64-bit time operators if the time value of **now** will exceed 2147483647 (the limit of 32-bit numbers). For example:

```
if { [gtTime $now 2us] } {
.
.
.
```

See "ModelSim Tcl time commands" (UM-325) for details on 64-bit time operators.

B - ModelSim shortcuts

Appendix contents

Wave window mouse and keyboard shortcuts UM-356
List window keyboard shortcuts UM-357
Command shortcuts UM-358
Command history shortcuts UM-358
Mouse and keyboard shortcuts in Main and Source windows . . . UM-359
Right mouse button UM-360

This appendix is a collection of the keyboard and command shortcuts available in the ModelSim GUI.

Wave window mouse and keyboard shortcuts

The following mouse actions and keystrokes can be used in the Wave window.

Mouse action	Result
< control - left-button - drag down and right>[a]	zoom area (in)
< control - left-button - drag up and right>	zoom out
< control - left-button - drag up and left>	zoom fit
<left-button - drag> (Select mode) < middle-button - drag> (Zoom mode)	moves closest cursor
< control - left-button - click on a scroll arrow >	scrolls window to very top or bottom(vertical scroll) or far left or right (horizontal scroll)

a. If you enter zoom mode by selecting **View > Mouse Mode > Zoom Mode**, you do not need to hold down the <Ctrl> key.

Keystroke	Action
i I or +	zoom in (mouse pointer must be over the the cursor or waveform panes)
o O or -	zoom out (mouse pointer must be over the the cursor or waveform panes)
f or F	zoom full (mouse pointer must be over the the cursor or waveform panes)
l or L	zoom last (mouse pointer must be over the the cursor or waveform panes)
r or R	zoom range (mouse pointer must be over the the cursor or waveform panes)
<up arrow>/ <down arrow>	with mouse over waveform pane, scrolls entire window up/ down one line; with mouse over pathname or values pane, scrolls highlight up/down one line
<left arrow>	scroll pathname, values, or waveform pane left
<right arrow>	scroll pathname, values, or waveform pane right
<page up>	scroll waveform pane up by a page
<page down>	scroll waveform pane down by a page

Keystroke	Action
<tab>	search forward (right) to the next transition on the selected signal - finds the next edge
<shift-tab>	search backward (left) to the previous transition on the selected signal - finds the previous edge
<control-f>	open the find dialog box; searches within the specified field in the pathname pane for text strings
<control-left arrow>	scroll pathname, values, or waveform pane left by a page
<control-right arrow>	scroll pathname, values, or waveform pane right by a page

List window keyboard shortcuts

Using the following keys when the mouse cursor is within the List window will cause the indicated actions:

Key	Action
<left arrow>	scroll listing left (selects and highlights the item to the left of the currently selected item)
<right arrow>	scroll listing right (selects and highlights the item to the right of the currently selected item)
<up arrow>	scroll listing up
<down arrow>	scroll listing down
<page up> <control-up arrow>	scroll listing up by page
<page down> <control-down arrow>	scroll listing down by page
<tab>	searches forward (down) to the next transition on the selected signal
<shift-tab>	searches backward (up) to the previous transition on the selected signal (does not function on HP workstations)
<shift-left arrow> <shift-right arrow>	extends selection left/right
<control-f>	opens the Find dialog box to find the specified item label within the list display

Command shortcuts

- You may abbreviate command syntax, but there's a catch — the minimum number of characters required to execute a command are those that make it unique. Remember, as we add new commands some of the old shortcuts may not work.
- Multiple commands may be entered on one line if they are separated by semi-colons (;). For example:

```
ModelSim> vlog -nodebug=ports level3.v level2.v ; vlog -nodebug top.v
```

  The return value of the last function executed is the only one printed to the transcript. This may cause some unexpected behavior in certain circumstances. Consider this example:

```
vsim -c -do "run 20 ; simstats ; quit -f" top
```

  You probably expect the **simstats** results to display in the Transcript window, but they will not, because the last command is **quit -f**. To see the return values of intermediate commands, you must explicitly print the results. For example:

```
vsim -do "run 20 ; echo [simstats]; quit -f" -c top
```

Command history shortcuts

The simulator command history may be reviewed, or commands may be reused, with these shortcuts at the ModelSim/VSIM prompt:

Shortcut	Description
up and down arrows	scrolls through the command history with the keyboard arrows
click on prompt	left-click once on a previous ModelSim or VSIM prompt in the transcript to copy the command typed at that prompt to the active cursor
`history`	shows the last few commands (up to 50 are kept)

Mouse and keyboard shortcuts in Main and Source windows

The following mouse actions and special keystrokes can be used to edit commands in the entry region of the Main window. They can also be used in editing the file displayed in the Source window and all Notepad windows (enter the **notepad** command within ModelSim to open the Notepad editor).

Keystrokes	Result
< left | right - arrow >	move the cursor left | right one character
< up | down - arrow >	scroll through command history (in Source window, move cursor one line up | down)
< control > < left | right - arrow >	move cursor left | right one word
< shift > < left | right | up | down - arrow >	extend selection of text
< control > < shift > < left | right - arrow >	extend selection of text by word
< up | down - arrow >	scroll through command history (in Source window, moves cursor one line up | down)
< control > < up | down >	move cursor up | down one paragraph
< alt >	activate or inactivate menu bar mode
< alt > < F4 >	close active window
< backspace >	delete character to the left
< home >	move cursor to the beginning of the line
< end >	move cursor to the end of the line
< control > < home >	move cursor to the beginning of the text
< control > < end >	move cursor to the end of the text
< esc >	cancel
< control - a >	select the entire content of the widget
< control - c >	copy the selection
< control - f >	find
< F3 >	find next
< control - k >	delete from the cursor to the end of the line
< control - s >	save
< control - t >	reverse the order of the two characters to the right of the cursor

Keystrokes	Result
< control - u >	delete line
< control - v >	paste from the clipboard
< control - x >	cut the selection
< F8 >	search for the most recent command that matches the characters typed
< F9 >	run simulation
< F10 >	continue simulation
< F11 >	single-step
< F12 >	step-over

The Main window allows insertions or pastes only after the prompt; therefore, you don't need to set the cursor when copying strings to the command line.

Right mouse button

The right mouse button provides shortcut menus in the most windows. See *Chapter 7 - Graphic interface* for menu descriptions.

C - ModelSim messages

Appendix contents

ModelSim message system UM-362
Message format UM-362
Getting more information UM-362

Suppressing warning messages UM-363
Suppressing VCOM warning messages UM-363
Suppressing VLOG warning messages UM-363
Suppressing VSIM warning messages UM-363

Exit codes UM-364

Miscellaneous messages UM-366
Empty port name warning UM-366
Lock message UM-366
Metavalue detected warning UM-366
Sensitivity list warning UM-367
Tcl Initialization error 2 . . . . . UM-367
Too few port connections UM-368
VSIM license lost UM-369

This appendix documents various status and warning messages that are produced by ModelSim.

ModelSim message system

The ModelSim message system helps you identify and troubleshoot problems while using the application. The messages display in a standard format in the Main window transcript. Accordingly, you can also access them from a saved transcript file (see "Saving the Main window transcript file" (UM-139) for more details).

Message format

The format for the messages is:

```
** <SEVERITY LEVEL>: ([<Tool>[-<Group>]]-<MsgNum>) <Message>
```

SEVERITY LEVEL may be one of the following:

severity level	meaning
Note	This is an informational message.
Warning	There may be a problem that will affect the accuracy of your results.
Error	The tool cannot complete the operation.
Fatal	The tool cannot complete execution.
INTERNAL ERROR	This is an unexpected error that should be reported to support@model.com.

Tool indicates which ModelSim tool was being executed when the message was generated. For example tool could be **vcom**, **vdel**, **vsim**, etc.

Group indicates the topic to which the problem is related. For example group could be FLI, PLI, VCD, etc.

Example

```
# ** Error: (vsim-PLI-3071) ./src/19/testfile(77): $fdumplimit : Too few
arguments.
```

Getting more information

Each message is identified by a unique MsgNum id. You can access additional information about a message using the unique id and the **verror** (CR-153) command. For example:

```
% verror 3071
Message # 3071:
Not enough arguments are being passed to the specified system task or
function.
```

Suppressing warning messages

You can suppress some warning messages. For example, you may receive warning messages about unbound components about which you are not concerned.

Suppressing VCOM warning messages

Use the `-nowarn <number>` argument to **vcom** (CR-145) to suppress a specific warning message. For example:

```
vcom -nowarn 1
```

Suppresses unbound component warning messages.

Alternatively, warnings may be disabled for all compiles via the *modelsim.ini* file (see "[vcom] VHDL compiler control variables" (UM-342)).

The warning message numbers are:

```
1 = unbound component
2 = process without a wait statement
3 = null range
4 = no space in time literal
5 = multiple drivers on unresolved signal
6 = compliance checks
7 = optimization messages
```

Suppressing VLOG warning messages

Use the `+nowarn<CODE>` argument to **vlog** (CR-181) to suppress a specific warning message. Warnings that can be disabled include the <CODE> name in square brackets in the warning message. For example:

```
vlog +nowarnDECAY
```

Suppresses decay warning messages.

Suppressing VSIM warning messages

Use the `+nowarn<CODE>` argument to **vsim** (CR-189) to suppress a specific warning message. Warnings that can be disabled include the <CODE> name in square brackets in the warning message. For example:

```
vlog +nowarnTFMPC
```

Suppresses warning messages about too few port connections.

Exit codes

The table below describes exit codes used by ModelSim tools.

Exit code	Description
0	Normal (non-error) return
1	Incorrect invocation of tool
2	Previous errors prevent continuing
3	Cannot create a system process (execv, fork, spawn, etc.)
4	Licensing problem
5	Cannot create/open/find/read/write a design library
6	Cannot create/open/find/read/write a design unit
7	Cannot open/read/write/dup a file (open, lseek, write, mmap, munmap, fopen, fdopen, fread, dup2, etc.)
8	File is corrupted or incorrect type, version, or format of file
9	Memory allocation error
10	General language semantics error
11	General language syntax error
12	Problem during load or elaboration
13	Problem during restore
14	Problem during refresh
15	Communication problem (Cannot create/read/write/close pipe/socket)
16	Version incompatibility
19	License manager not found/unreadable/unexecutable (vlm/mgvlm)
42	Lost license
43	License read/write failure
44	Modeltech daemon license checkout failure #44
45	Modeltech daemon license checkout failure #45
90	Assertion failure (SEVERITY_QUIT)
99	Unexpected error in tool
202	Interrupt (SIGINT)
204	Illegal instruction (SIGILL)

Exit code	Description
205	Trace trap (SIGTRAP)
206	Abort (SIGABRT)
208	Floating point exception (SIGFPE)
210	Bus error (SIGBUS)
211	Segmentation violation (SIGSEGV)
213	Write on a pipe with no reader (SIGPIPE)
214	Alarm clock (SIGALRM)
215	Software termination signal from kill (SIGTERM)
216	User-defined signal 1 (SIGUSR1)
217	User-defined signal 2 (SIGUSR2)
218	Child status change (SIGCHLD)
230	Exceeded CPU limit (SIGXCPU)
231	Exceeded file size limit (SIGXFSZ)

Miscellaneous messages

This section describes miscellaneous messages which may be associated with ModelSim.

Empty port name warning

Message text

```
# WARNING[8]: <path/file_name>:
empty port name in port list.
```

Meaning

ModelSim reports these warnings if you use the **-lint** argument to **vlog** (CR-181). It reports the warning for any NULL module ports.

Suggested action

If you wish to ignore this warning, do not use the **-lint** argument.

Lock message

Message text

```
waiting for lock by user@user. Lockfile is <library_path>/_lock
```

Meaning

The *_lock* file is created in a library when you begin a compilation into that library, and it is removed when the compilation completes. This prevents simultaneous updates to the library. If a previous compile did not terminate properly, ModelSim may fail to remove the *_lock* file.

Suggested action

Manually remove the *_lock* file after making sure that no one else is actually using that library.

Metavalue detected warning

Message text

```
Warning: NUMERIC_STD.">": metavalue detected, returning FALSE
```

Meaning

This warning is an assertion being issued by the IEEE **numeric_std** package. It indicates that there is an 'X' in the comparison.

Suggested action

The message does not indicate which comparison is reporting the problem since the assertion is coming from a standard package. To track the problem, note the time the warning occurs, restart the simulation, and run to one time unit before the noted time. At this point, start stepping the simulator until the warning appears. The location of the blue

arrow in the source window will be pointing at the line following the line with the comparison.

These messages can be turned off by setting the **NumericStdNoWarnings** variable to 1 from the command line or in the *modelsim.ini* file.

Sensitivity list warning

Message text

```
signal is read by the process but is not in the sensitivity list
```

Meaning

ModelSim outputs this message when you use the **-check_synthesis** argument to **vcom** (CR-145). It reports the warning for any signal that is read by the process but is not in the sensitivity list.

Suggested action

There are cases where you may purposely omit signals from the sensitivity list even though they are read by the process. For example, in a strictly sequential process, you may prefer to include only the clock and reset in the sensitivity list because it would be a design error if any other signal triggered the process. In such cases, you're only option as of version 5.7 is to not use the **-check_synthesis** argument. A more robust implementation of the argument may be added to a future version.

Tcl Initialization error 2

Message text

```
Tcl_Init Error 2 : Can't find a usable Init.tcl in the following directories :
   ./../tcl/tcl8.3 .
```

Meaning

This message typically occurs when the base file was not included in a Unix installation. When you install ModelSim, you need to download and install 3 files from the ftp site. These files are:

- modeltech-base.tar.gz
- modeltech-docs.tar.gz
- modeltech-<platform>.exe.gz

If you install only the <platform> file, you will not get the Tcl files that are located in the base file.

This message could also occur if the file or directory was deleted or corrupted.

Suggested action

Reinstall ModelSim with all three files.

Too few port connections

Message text

```
# ** Warning (vsim-3017): foo.v(1422): [TFMPC] - Too few port connections.
Expected # 2, found 1. Region: /foo/tb
```

Meaning

This warning occurs when an instantiation has fewer port connections than the corresponding module definition. The warning doesn't necessarily mean anything is wrong; it is legal in Verilog to have an instantiation that doesn't connect all of the pins. However, someone that expects all pins to be connected would like to see such a warning.

Here are some examples of legal instantiations that will and will not cause the warning message.

Module definition:

```
module foo (a, b, c, d);
```

Instantiation that does not connect all pins but will not produce the warning:

`foo inst1(e, f, g, );` – positional association

`foo inst1(.a(e), .b(f), .c(g), .d());` – named association

Instantiation that does not connect all pins but will produce the warning:

`foo inst1(e, f, g);` – positional association

`foo inst1(.a(e), .b(f), .c(g));` – named association

Any instantiation above will leave pin *d* unconnected but the first example has a placeholder for the connection. Here's another example:

```
foo inst1(e, , g, h);
```

```
foo inst1(.a(e), .b(), .c(g), .d(h));
```

Suggested actions

- Check that there is not an extra comma at the end of the port list. (e.g., model(a,b,)). The extra comma is legal Verilog and implies that there is a third port connection that is unnamed.
- If you are purposefully leaving pins unconnected, you can disable these messages using the **+nowarnTFMPC** argument to vsim.

VSIM license lost

Message text

```
Console output:
Signal 0 caught... Closing vsim vlm child.
vsim is exiting with code 4
FATAL ERROR in license manager

transcript/vsim output:
# ** Error: VSIM license lost; attempting to re-establish.
#    Time: 5027 ns  Iteration: 2
# ** Fatal: Unable to kill and restart license process.
#    Time: 5027 ns  Iteration: 2
```

Meaning

ModelSim queries the license server for a license at regular intervals. Usually these "License Lost" error messages indicate that network traffic is high, and communication with the license server times out.

Suggested action

Anything you can do to improve network communication with the license server will probably solve or decrease the frequency of this problem.

D - System initialization

Appendix contents

Files accessed during startup UM-372

Environment variables accessed during startup UM-373

Initialization sequence UM-374

ModelSim goes through numerous steps as it initializes the system during startup. It accesses various files and environment variables to determine library mappings, configure the GUI, check licensing, and so forth.

Files accessed during startup

The table below describes the files that are read during startup. They are listed in the order in which they are accessed.

File	Purpose
modelsim.ini	contains initial tool settings; see "Preference variables located in INI files" (UM-341) for specific details on the *modelsim.ini* file
location map file	used by ModelSim tools to find source files based on easily reallocated "soft" paths; default file name is mgc_location_map
pref.tcl	contains defaults for fonts, colors, prompts, window positions, and other simulator window characteristics; see "Preference variables located in Tcl files" (UM-352) for specific details on the *pref.tcl* file
modelsim.tcl	contains user-customized settings for fonts, colors, prompts, window positions, and other simulator window characteristics; see "Preference variables located in Tcl files" (UM-352) for specific details on the *modelsim.tcl* file
<project_name>.mpf	if available, loads last project file which is specified in the registry (Windows); see "What are projects?" (UM-18) for details on project settings

Environment variables accessed during startup

The table below describes the environment variables that are read during startup. They are listed in the order in which they are accessed. For more information on environment variables, see "Environment variables" (UM-337).

Environment variable	Purpose
MODEL_TECH	set by ModelSim to the directory in which the binary executables reside (e.g., *../modeltech/<platform>/*)
MODEL_TECH_OVERRIDE	provides an alternative directory for the binary executables; MODEL_TECH is set to this path
MODELSIM	identifies path to the *modelsim.ini* file
MGC_WD	identifies the Mentor Graphics working directory
MGC_LOCATION_MAP	identifies the path to the location map file; set by ModelSim if not defined
MODEL_TECH_TCL	identifies the path to all Tcl libraries installed with ModelSim
HOME	identifies your login directory (UNIX only)
MGC_HOME	identifies the path to the MGC tool suite
TCL_LIBRARY	identifies the path to the Tcl library; set by ModelSim to the same path as MODEL_TECH_TCL; must point to libraries supplied by Model Technology
TK_LIBRARY	identifies the path to the Tk library; set by ModelSim to the same path as MODEL_TECH_TCL; must point to libraries supplied by Model Technology
ITCL_LIBRARY	identifies the path to the [incr]Tcl library; set by ModelSim to the same path as MODEL_TECH_TCL; must point to libraries supplied by Model Technology
ITK_LIBRARY	identifies the path to the [incr]Tk library; set by ModelSim to the same path as MODEL_TECH_TCL; must point to libraries supplied by Model Technology
VSIM_LIBRARY	identifies the path to the Tcl files that are used by ModelSim; set by ModelSim to the same path as MODEL_TECH_TCL; must point to libraries supplied by Model Technology
MTI_COSIM_TRACE	creates an *mti_trace_cosim* file containing debugging information about FLI/PLI/VPI function calls; set to any value before invoking the simulator.
MTI_LIB_DIR	identifies the path to all Tcl libraries installed with ModelSim
MODELSIM_TCL	identifies the path to the *modelsim.tcl* file; this environment variable can be a list of file pathnames, separated by semicolons (Windows)

Initialization sequence

The following list describes in detail ModelSim's initialization sequence. The sequence includes a number of conditional structures, the results of which are determined by the existence of certain files and the current settings of environment variables.

In the steps below, names in uppercase denote environment variables (except MTI_LIB_DIR which is a Tcl variable). Instances of *$(NAME)* denote paths that are determined by an environment variable (except *$(MTI_LIB_DIR)* which is determined by a Tcl variable).

1 Determines the path to the executable directory (../modeltech/<platform>/). Sets MODEL_TECH to this path, *unless* MODEL_TECH_OVERRIDE exists, in which case MODEL_TECH is set to the same value as MODEL_TECH_OVERRIDE.

2 Finds the *modelsim.ini* file by evaluating the following conditions:

- use MODELSIM if it exists; else
- use *$(MGC_WD)/modelsim.ini*; else
- use *./modelsim.ini*; else
- use *$(MODEL_TECH)/modelsim.ini*; else
- use *$(MODEL_TECH)/../modelsim.ini*; else
- use *$(MGC_HOME)/lib/modelsim.ini*; else
- set path to *./modelsim.ini* even though the file doesn't exist

3 Finds the location map file by evaluating the following conditions:

- use MGC_LOCATION_MAP if it exists (if this variable is set to "no_map", ModelSim skips initialization of the location map); else
- use *mgc_location_map* if it exists; else
- use *$(HOME)/mgc/mgc_location_map*; else
- use *$(HOME)/mgc_location_map*; else
- use *$(MGC_HOME)/etc/mgc_location_map*; else
- use *$(MGC_HOME)/shared/etc/mgc_location_map*; else
- use *$(MODEL_TECH)/mgc_location_map*; else
- use *$(MODEL_TECH)/../mgc_location_map*; else
- use no map

4 Reads various variables from the [vsim] section of the *modelsim.ini* file. See "[vsim] simulator control variables" (UM-344) for more details.

5 Parses any command line arguments that were included when you started ModelSim and reports any problems.

6 Defines the following environment variables:

- use MODEL_TECH_TCL if it exists; else

- set MODEL_TECH_TCL=*$(MODEL_TECH)/../tcl*
- set TCL_LIBRARY=*$(MODEL_TECH_TCL)/tcl8.3*
- set TK_LIBRARY=*$(MODEL_TECH_TCL)/tk8.3*
- set ITCL_LIBRARY=*$(MODEL_TECH_TCL)/itcl3.0*
- set ITK_LIBRARY=*$(MODEL_TECH_TCL)/itk3.0*
- set VSIM_LIBRARY=*$(MODEL_TECH_TCL)/vsim*

7 Initializes the simulator's Tcl interpreter.

8 Checks for a valid license (a license is not checked out unless specified by a *modelsim.ini* setting or command line option).

The next four steps relate to initializing the graphical user interface.

9 Sets Tcl variable "MTI_LIB_DIR"=MODEL_TECH_TCL

10 Loads *$(MTI_LIB_DIR)/pref.tcl*.

11 Finds the *modelsim.tcl* file by evaluating the following conditions:

- use MODELSIM_TCL environment variable if it exists (if MODELSIM_TCL is a list of files, each file is loaded in the order that it appears in the list); else
- use *./modelsim.tcl*; else
- use *$(HOME)/modelsim.tcl* if it exists

12 Loads last working directory, project file, and printer defaults from the registry (Windows).

That completes the initialization sequence. Also note the following about the *modelsim.ini* file:

- When you change the working directory within ModelSim, the tool reads the [library], [vcom], and [vlog] sections of the local *modelsim.ini* file. When you make changes in the compiler options dialog or use the **vmap** command, the tool updates the appropriate sections of the file.
- The *pref.tcl* file references the default .ini file via the [GetPrivateProfileString] Tcl command. The .ini file that is read will be the default file defined at the time *pref.tcl* is loaded.

E - Tips and techniques

Appendix contents

Setting up libraries for group use UM-379
Using a DO file to test for assertions UM-380
Locating assertion warnings UM-380
Sampling signals at a clock change UM-381
Configuring a List trigger with Expression Builder UM-382
Converting signal values to strings UM-384
Converting an integer into a bit_vector UM-385
Referencing source files with location maps UM-387
Performance affected by scheduled events being cancelled . . . UM-389
Modeling memory in VHDL UM-390

This appendix contains various tips and techniques collected from several parts of the manual and from answers to questions received by tech support.

Running command-line and batch-mode simulations

The typical method of running ModelSim is interactive: you push buttons and/or pull down menus in a series of windows in the GUI (graphic user interface). But there are really three specific modes of ModelSim operation: GUI, command line, and batch. Here are their characteristics:

- **GUI mode**
  This is the usual interactive mode; it has graphical windows, push-buttons, menus, and a command line in the text window. This is the default mode.
- **Command-line mode - running vsim.exe**
  This an operational mode that has only an interactive command line; no interactive windows are opened. To run **vsim** in this manner, invoke it with the **-c** option as the first argument from the DOS prompt in Windows.

  The resulting transcript file is created in such a way that the transcript can be re-executed without change if you desire. Everything except the explicit commands you enter will begin with a leading comment character (#).
- **Batch mode - running vsim.exe**
  Batch mode is an operational mode that provides neither an interactive command line, nor interactive windows.

  In a Windows environment, **vsim** is run from a Windows command prompt and standard input and output are re-directed to and from files. An example of the "here-document" technique is:

  ```
  C:\modeltech> vsim ent arch <infile >outfile
  ```

  where *infile* contains:

  ```
  force reset 0
  force clk 0, 0 1 50 -rep 100
  run 10000
  ```

Saving and viewing waveforms in batch mode

You can run **vsim** as a batch job and view the resulting waveforms later.

1. When you invoke **vsim** the first time, use the **-wlf** option to rename the wave log format (WLF) file, and redirect stdin to invoke the batch mode. The command should look like this:

   ```
   vsim -wlf wavesav1.wlf counter < command.do
   ```

   Within your *command.do* file, use the **log** command (CR-87) to save the waveforms you want to look at later, run the simulation, and quit.

   When **vsim** runs in batch mode, it does not write to the screen, and can be run in the background.

2. When you return to work the next day after running several batch jobs, you can start up **vsim** in its viewing mode with this command and the appropriate *.wlf* files:

   ```
   vsim -view wavesav1.wlf
   ```

Now you will be able to use the Waveform and List windows normally.

Setting up libraries for group use

By adding an “others” clause to your *modelsim.ini* file, you can have a hierarchy of library mappings. If the ModelSim tools don’t find a mapping in the *modelsim.ini* file, then they will search the library section of the initialization file specified by the “others” clause. For example:

```
[library]
asic_lib = /cae/asic_lib
work = my_work
others = /usr/modeltech/modelsim.ini
```

Using a DO file to test for assertions

You can use the **onbreak** command (CR-98) in a DO file to invoke commands upon the occurrence of a simulation breakpoint. Assertions are treated as breakpoints if the severity level is greater than or equal to the current BreakOnAssertion variable setting (see "[vsim] simulator control variables" (UM-344)). By default a severity level of failure or above causes a breakpoint; a severity level of error or below does not.

Here is an example of how the onbreak command might be used to test for an assertion:

```
set broken 0
onbreak {
  set broken 1
  resume
}
run -all
if { $broken } {
  puts "failure"
} else {
  puts "success"
}
```

Locating assertion warnings

You may receive assertion messages that don't contain file and line numbers. For example:

```
# ** Warning: NUMERIC_STD.TO_UNSIGNED: vector truncated
#    Time: 0 ns  Iteration: 0  Instance: /core_tb
# ** Warning: NUMERIC_STD.TO_INTEGER: metavalue detected, returning 0
#    Time: 0 ns  Iteration: 0  Instance: /core_tb
```

Set the BreakOnAssertion (UM-345) value to break on warnings. Any assertion warnings will be treated as breakpoints, and you'll be able to see the file and line number in the Source window.

The value you specify determines what severity level causes a simulation break (0 = note, 1 = warning, 2 = error, 3 = failure, 4 = fatal). You can specify this in the *modelsim.ini* file or from the GUI by selecting **Simulate > Simulation Options** (Main window) and selecting the Assertions tab.

Sampling signals at a clock change

You can do this easily using the **add list** command (CR-32) with the **-notrigger** argument. -notrigger disables triggering the display on the specified signals. For example:

```
add list clk -notrigger a b c
```

When you run the simulation, List window entries for *clk*, *a*, *b*, and *c* appear only when *clk* changes.

If you want to display on rising edges only, you have two options:

1. Turn off the List window triggering on the clock signal, and then define a repeating strobe for the list window

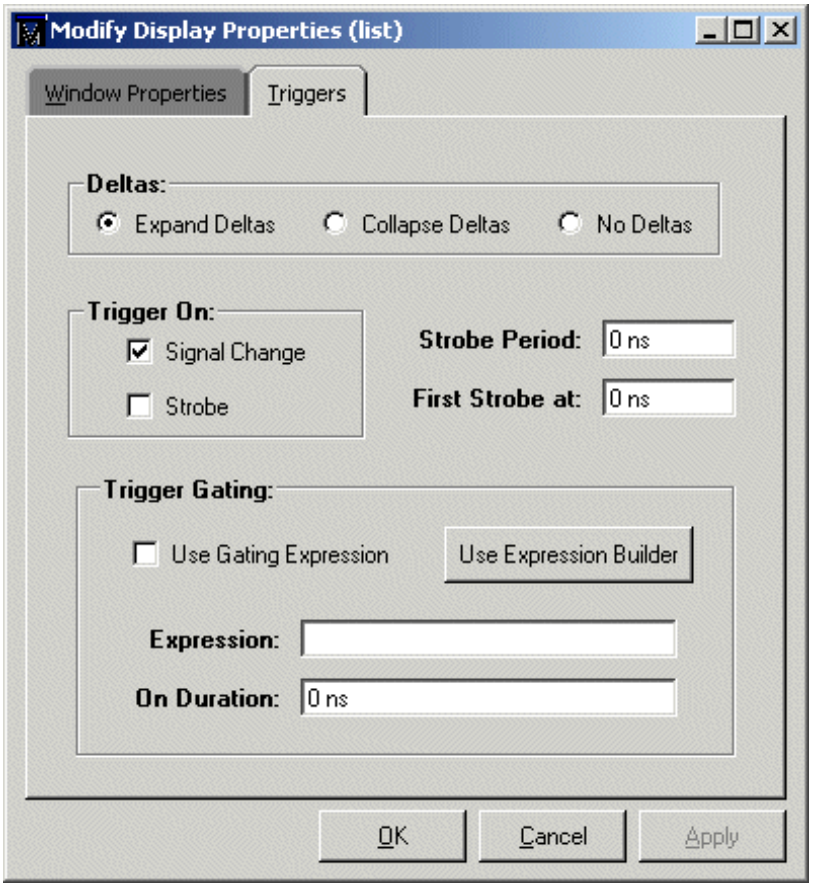
2. Define a "gating expression" for the List window that requires the clock to be in a specified state. See "Configuring a List trigger with Expression Builder" (UM-382).

Configuring a List trigger with Expression Builder

This example shows you how to set a List window trigger based on a gating expression created with the ModelSim Expression Builder.

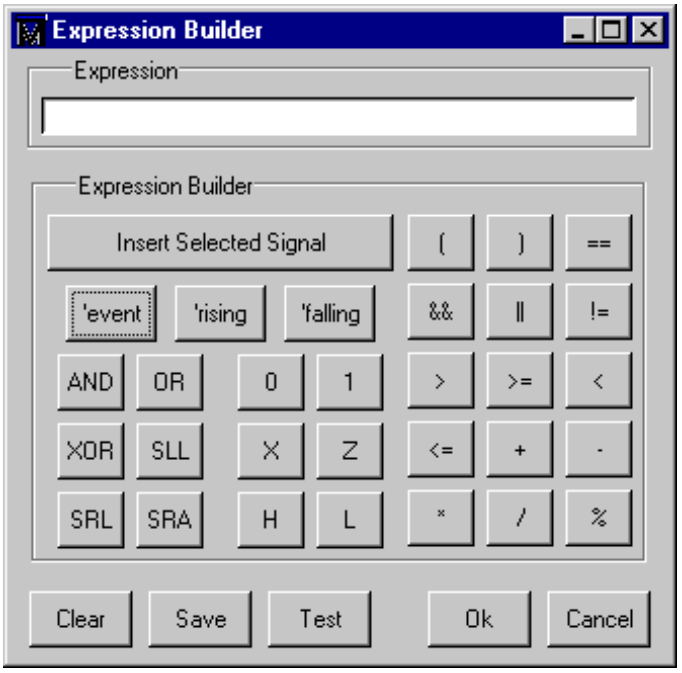
If you want to look at a set of signal values ONLY during the simulation cycles during which an enable signal rises, you would need to use the List window Trigger Gating feature. The gating feature suppresses all display lines except those for which a specified gating function evaluates to true.

Select **Tools** > **Window Preferences** (List window) to access the Triggers tab.



Check the **Trigger Gating: Use Gating Expression** check box. Then click on **Use Expression Builder**. Select the signal in the List window that you want to be the enable

signal by clicking on its name in the header area of the List window. Then click **Insert Selected Signal** and **'rising** in the Expression Builder.



Click OK to close the Expression Builder. You should see the name of the signal plus "'rising" added to the Expression entry box of the Modify Display Properties dialog box. (Leave the **On Duration** field zero for now.) Click the **OK** button.

If you already have simulation data in the List window, the display should immediately switch to showing only those cycles for which the gating signal is rising. If that isn't quite what you want, you can go back to the expression builder and play with it until you get it the way you want it.

If you want the enable signal to work like a "One-Shot" that would display all values for the next, say 10 ns, after the rising edge of enable, then set the **On Duration** value to **10 ns**. Otherwise, leave it at zero, and select **Apply** again. When everything is correct, click **OK** to close the Modify Display Properties dialog box.

When you save the List window configuration, the list gating parameters will be saved as well, and can be set up again by reading in that macro. You can take a look at the macro to see how the gating can be set up using macro commands.

Converting signal values to strings

You may want to display certain signal values as strings. For example, rather than displaying the value 0, you may want to display the string "idle." The **virtual type** command (CR-178) allows you to do this.

The virtual type command creates a new enumerated type, known only by the GUI. The steps for using the command are as follows:

1 Define a virtual type that contains the states:

```
virtual type { state0 state1 state2 state3} myState
```

2 Define a virtual function for translating the signal values to strings

```
virtual function {(mystate)mysignal} myConvertedSignal
```

3 Display the translated value

```
add wave myConvertedSignal
```

When myConvertedSignal is displayed in the Wave, List or Signals window, the string "state0" will appear when mysignal == 0, "state1" when mysignal == 1, "state2" when mysignal == 2, etc.

See the **virtual type** command (CR-178) in the *ModelSim Command Reference* for further details.

Converting an integer into a bit_vector

The following code demonstrates how to convert an integer into a bit_vector.

```
library ieee;
use ieee.numeric_bit.ALL;

entity test is
end test;

architecture only of test is
  signal s1 : bit_vector(7 downto 0);
  signal int : integer := 45;
begin
  p:process
  begin
    wait for 10 ns;
    s1 <= bit_vector(to_signed(int,8));
  end process p;
end only;
```

Detecting infinite zero-delay loops

Simulations use steps that advance simulated time, and steps that do not advance simulated time. Steps that do not advance simulated time are called "delta cycles’ or simply ’deltas’. Deltas are used when signal assignments are made with zero time delay (see "Delta delays" (UM-53)for more information).

If a large number of deltas occur without advancing time, it is usually a symptom of an infinite zero-delay loop in the design. In order to detect the presence of these loops, ModelSim defines a limit, the “iteration limit", on the number of successive deltas that can occur. When the iteration limit is exceeded, **vsim** stops the simulation and gives a warning message.

The iteration limit default value is1000. If you receive an iteration limit warning, first increase the iteration limit and try to continue simulation. You can set the iteration limit from the **Simulate > Simulation Options** menu, or by modifying the *modelsim.ini* file.See for more information on modifying the *modelsim.ini* file.

If the problem persists, look for zero-delay loops. Run the simulation and look at the source code when the error occurs. Use the step button to step through the code and see which signals or variables are continuously oscillating. Two common causes are a loop that has no exit, or a series of gates with zero delay where the outputs are connected back to the inputs.

Referencing source files with location maps

Pathnames to source files are recorded in libraries by storing the working directory from which the compile is invoked and the pathname to the file as specified in the invocation of the compiler. The pathname may be either a complete pathname or a relative pathname.

ModelSim tools that reference source files from the library locate a source file as follows:

- If the pathname stored in the library is complete, then this is the path used to reference the file.
- If the pathname is relative, then the tool looks for the file relative to the current working directory. If this file does not exist, then the path relative to the working directory stored in the library is used.

This method of referencing source files generally works fine if the libraries are created and used on a single system. However, when multiple systems access a library across a network the physical pathnames are not always the same and the source file reference rules do not always work.

Using location mapping

Location maps are used to replace prefixes of physical pathnames in the library with environment variables. The location map defines a mapping between physical pathname prefixes and environment variables.

ModelSim tools open the location map file on invocation if the MGC_LOCATION_MAP (UM-337) environment variable is set. If MGC_LOCATION_MAP is not set, ModelSim will look for a file named *"mgc_location_map"* in the following locations, in order:

- the current directory
- your home directory
- the directory containing the ModelSim binaries
- the ModelSim installation directory

Use these two steps to map your files:

1. Set the environment variable MGC_LOCATION_MAP to the path to your location map file.

2. Specify the mappings from physical pathnames to logical pathnames:

```
$SRC
/home/vhdl/src
/usr/vhdl/src

$IEEE
/usr/modeltech/ieee
```

Pathname syntax

The logical pathnames must begin with *$* and the physical pathnames must begin with */*. The logical pathname is followed by one or more equivalent physical pathnames. Physical pathnames are equivalent if they refer to the same physical directory (they just have different pathnames on different systems).

How location mapping works

When a pathname is stored, an attempt is made to map the physical pathname to a path relative to a logical pathname. This is done by searching the location map file for the first physical pathname that is a prefix to the pathname in question. The logical pathname is then substituted for the prefix. For example, "/usr/vhdl/src/test.vhd" is mapped to "$SRC/test.vhd". If a mapping can be made to a logical pathname, then this is the pathname that is saved. The path to a source file entry for a design unit in a library is a good example of a typical mapping.

For mapping from a logical pathname back to the physical pathname, ModelSim expects an environment variable to be set for each logical pathname (with the same name). ModelSim reads the location map file when a tool is invoked. If the environment variables corresponding to logical pathnames have not been set in your shell, ModelSim sets the variables to the first physical pathname following the logical pathname in the location map. For example, if you don't set the SRC environment variable, ModelSim will automatically set it to "/home/vhdl/src".

Mapping with Tcl variables

Two Tcl variables may also be used to specify alternative source-file paths; SourceDir and SourceMap. See http://www.model.com/resources/pref_variables/frameset.htm.

Performance affected by scheduled events being cancelled

Performance will suffer if events are scheduled far into the future but then cancelled before they take effect. This situation will act like a memory leak and slow down simulation.

In VHDL this situation can occur several ways. The most common are waits with time-out clauses and projected waveforms in signal assignments.

The following code shows a wait with a time-out:

```
signals synch : bit := '0';
...
p: process
begin
    wait for 10 ms until synch = 1;
end process;

synch <= not synch after 10 ns;
```

At time 0, process *p* makes an event for time 10ms. When *synch* goes to 1 at 10 ns, the event at 10 ms is marked as cancelled but not deleted, and a new event is scheduled at 10ms + 10ns. The cancelled events are not reclaimed until time 10ms is reached and the cancelled event is processed. As a result there will be 500000 (10ms/20ns) cancelled but undeleted events. Once 10ms is reached, memory will no longer increase because the simulator will be reclaiming events as fast as they are added.

For projected waveforms the following would behave the same way:

```
signals synch : bit := '0';
...
p: process(synch)
begin
   output <= '0', '1' after 10ms;
end process;

synch <= not synch after 10 ns;
```

Modeling memory in VHDL

As a VHDL user, you might be tempted to model a memory using signals. Two common simulator problems are the likely result:

- You may get a "memory allocation error" message, which typically means the simulator ran out of memory and failed to allocate enough storage.
- Or, you may get very long load, elaboration or run times.

These problems are usually explained by the fact that signals consume a substantial amount of memory (many dozens of bytes per bit), all of which needs to be loaded or initialized before your simulation starts.

A simple alternative implementation provides some excellent performance benefits:

- storage required to model the memory can be reduced by 1-2 orders of magnitude
- startup and run times are reduced
- associated memory allocation errors are eliminated

The trick is to model memory using variables instead of signals.

In the example below, we illustrate three alternative architectures for entity "memory". Architecture "style_87_bad" uses a vhdl signal to store the ram data. Architecture "style_87" uses variables in the "memory" process, and architecture "style_93" uses variables in the architecture.

For large memories, architecture "style_87_bad" runs many times longer than the other two, and uses much more memory. This style should be avoided.

Both architectures "style_87" and "style_93" work with equal efficiently. You'll find some additional flexibility with the VHDL 1993 style, however, because the ram storage can be shared between multiple processes. For example, a second process is shown that initializes the memory; you could add other processes to create a multi-ported memory.

To implement this model, you will need functions that convert vectors to integers. To use it you will probably need to convert integers to vectors.

Example functions are provided below in package "conversions".

```
library ieee;
use ieee.std_logic_1164.all;
use work.conversions.all;

entity memory is
    generic(add_bits : integer := 12;
            data_bits : integer := 32);
    port(add_in : in std_ulogic_vector(add_bits-1 downto 0);
        data_in : in std_ulogic_vector(data_bits-1 downto 0);
        data_out : out std_ulogic_vector(data_bits-1 downto 0);
        cs, mwrite : in std_ulogic;
        do_init : in std_ulogic);
    subtype word is std_ulogic_vector(data_bits-1 downto 0);
    constant nwords : integer := 2 ** add_bits;
    type ram_type is array(0 to nwords-1) of word;
end;

architecture style_93 of memory is
        ------------------------------
        shared variable ram : ram_type;
        ------------------------------
```

```
begin
memory:
process (cs)
    variable address : natural;
    begin
        if rising_edge(cs) then
            address := sulv_to_natural(add_in);
            if (mwrite = '1') then
                 ram(address) := data_in;
            end if;
            data_out <= ram(address);
        end if;
    end process memory;
-- illustrates a second process using the shared variable
initialize:
process (do_init)
    variable address : natural;
    begin
        if rising_edge(do_init) then
            for address in 0 to nwords-1 loop
                ram(address) := data_in;
            end loop;
        end if;
    end process initialize;
end architecture style_93;

architecture style_87 of memory is
begin
memory:
process (cs)
    ----------------------
    variable ram : ram_type;
    ----------------------
    variable address : natural;
    begin
        if rising_edge(cs) then
            address := sulv_to_natural(add_in);
            if (mwrite = '1') then
                 ram(address) := data_in;
            end if;
            data_out <= ram(address);
        end if;
    end process;
end style_87;

architecture bad_style_87 of memory is
    ----------------------
    signal ram : ram_type;
    ----------------------
begin
memory:
process (cs)
    variable address : natural := 0;
    begin
        if rising_edge(cs) then
            address := sulv_to_natural(add_in);
            if (mwrite = '1') then
                ram(address) <= data_in;
                data_out <= data_in;
            else
                data_out <= ram(address);
```

```
            end if;
        end if;
    end process;
end bad_style_87;

------------------------------------------------------------
------------------------------------------------------------
library ieee;
use ieee.std_logic_1164.all;

package conversions is
    function sulv_to_natural(x : std_ulogic_vector) return
                natural;
    function natural_to_sulv(n, bits : natural) return
                std_ulogic_vector;
end conversions;

package body conversions is

    function sulv_to_natural(x : std_ulogic_vector) return
                natural is
        variable n : natural := 0;
        variable failure : boolean := false;
    begin
        assert (x'high - x'low + 1) <= 31
            report "Range of sulv_to_natural argument exceeds
                natural range"
            severity error;
        for i in x'range loop
            n := n * 2;
            case x(i) is
                when '1' | 'H' => n := n + 1;
                when '0' | 'L' => null;
                when others    => failure := true;
            end case;
        end loop;
        assert not failure
            report "sulv_to_natural cannot convert indefinite
                std_ulogic_vector"
            severity error;

        if failure then
            return 0;
        else
            return n;
        end if;
    end sulv_to_natural;

    function natural_to_sulv(n, bits : natural) return
                std_ulogic_vector is
        variable x : std_ulogic_vector(bits-1 downto 0) :=
                (others => '0');
        variable tempn : natural := n;
    begin
        for i in x'reverse_range loop
            if (tempn mod 2) = 1 then
                x(i) := '1';
            end if;
            tempn := tempn / 2;
        end loop;
        return x;
```

```
    end natural_to_sulv;

end conversions;
```

A B C D E F G H I J K L M N O P Q R S T U V W X Y Z

Index

CR = Command Reference, UM = User's Manual

Symbols

+typdelays CR-184
.so, shared object file
    loading PLI/VPI C applications UM-101
    loading PLI/VPI C++ applications UM-102
'hasX, hasX CR-19

Numerics

1076, IEEE Std UM-14
1364, IEEE Std UM-14, UM-68
64-bit time
    now variable UM-354
    Tcl time commands UM-325

A

abort command CR-30
absolute time, using @ CR-14
ACC routines UM-110
accelerated packages UM-47
add list command CR-32
add wave command CR-35
alias command CR-39
annotating interconnect delays, v2k_int_delays CR-200
architecture simulator state variable UM-353
archives
    described UM-38
archives, library CR-180
argc simulator state variable UM-353
arguments
    passing to a DO file UM-331
arithmetic package warnings, disabling UM-350
arrays
    indexes CR-10
    slices CR-10
AssertFile .ini file variable UM-344
AssertionFormat .ini file variable UM-344
AssertionFormatBreak .ini file variable UM-344
AssertionFormatError .ini file variable UM-344
AssertionFormatFail .ini file variable UM-345
AssertionFormatFatal .ini file variable UM-345
AssertionFormatNote .ini file variable UM-344
AssertionFormatWarning .ini file variable UM-344
assertions
    configuring from the GUI UM-255
    locating file and line number UM-380
    messages, turning off UM-350
    selecting severity that stops simulation UM-255
    setting format of messages UM-344
    testing for using a DO file UM-380
attributes, of signals, using in expressions CR-19

B

bad magic number error message UM-119
balloon dialog, toggling on/off UM-223
base (radix), specifying in List window UM-173
batch_mode command CR-40
batch-mode simulations UM-378
    halting CR-208
bd (breakpoint delete) command CR-41
binding, VHDL, default UM-45
blocking assignments UM-81
bookmark add wave command CR-42
bookmark delete wave command CR-43
bookmark goto wave command CR-44
bookmark list wave command CR-45
bookmarks UM-229
bp (breakpoint) command CR-46
break
    on assertion UM-255
    on signal value CR-205
    stop simulation run UM-146, UM-196
BreakOnAssertion .ini file variable UM-345
breakpoints
    conditional CR-205, UM-189
    continuing simulation after CR-114
    deleting CR-41, UM-197, UM-258
    listing CR-46
    setting CR-46, UM-197
    signal breakpoints (when statements) CR-205, UM-189
    Source window, viewing in UM-191
    time-based UM-189
        in when statements CR-209
.bsm file UM-165
buffered/unbuffered output UM-348
busses
    RTL-level, reconstructing UM-126
    user-defined CR-36, UM-174, UM-217

C

A B C D E F G H I J K L M N O P Q R S T U V W X Y Z

C applications
    compiling and linking UM-101
C++ applications
    compiling and linking UM-102
case choice, must be locally static CR-147
case sensitivity
    VHDL vs. Verilog CR-12
causality, tracing in Dataflow window UM-159
cd (change directory) command CR-49
cell libraries UM-87
cells
    hiding in Dataflow window UM-166, UM-167
change command CR-50
chasing X UM-160
-check_synthesis argument CR-145
CheckpointCompressMode .ini file variable UM-345
CheckSynthesis .ini file variable UM-342
clock change, sampling signals at UM-381
combining signals, user-defined bus CR-36, UM-174, UM-217
command history UM-143
CommandHistory .ini file variable UM-345
command-line mode UM-378
commands
    abort CR-30
    add list CR-32
    add wave CR-35
    alias CR-39
    batch_mode CR-40
    bd (breakpoint delete) CR-41
    bookmark add wave CR-42
    bookmark delete wave CR-43
    bookmark goto wave CR-44
    bookmark list wave CR-45
    bp (breakpoint) CR-46
    cd (change directory) CR-49
    change CR-50
    configure CR-51
    dataset alias CR-55
    dataset clear CR-56
    dataset close CR-57
    dataset info CR-58
    dataset list CR-59
    dataset open CR-60
    dataset rename CR-61, CR-62
    dataset snapshot CR-63
    delete CR-65
    describe CR-66
    disablebp CR-67
    do CR-68
    drivers CR-69
    dumplog64 CR-70
    echo CR-71
    edit CR-72
    enablebp CR-73
    environment CR-74
    examine CR-75
    exit CR-78
    find CR-79
    force CR-82
    graphic interface commands UM-267
    help CR-85
    history CR-86
    log CR-87
    lshift CR-89
    lsublist CR-90
    modelsim CR-91
    noforce CR-92
    nolog CR-93
    notation conventions CR-6
    notepad CR-95
    noview CR-96
    nowhen CR-97
    onbreak CR-98
    onElabError CR-99
    onerror CR-100
    pause CR-101
    printenv CR-102, CR-103
    pwd CR-105
    quietly CR-106
    quit CR-107
    radix CR-108
    report CR-109
    restart CR-111
    resume CR-113
    run CR-114
    searchlog CR-116
    shift CR-118
    show CR-119
    status CR-121
    step CR-122
    stop CR-123
    system UM-323
    tb (traceback) CR-124
    transcript CR-125
    TreeUpdate CR-217
    tssi2mti CR-126
    variables referenced in CR-13
    vcd add CR-127
    vcd checkpoint CR-128
    vcd comment CR-129
    vcd dumpports CR-130

A B C D E F G H I J K L M N O P Q R S T U V W X Y Z

vcd dumpportsall CR-131
vcd dumpportsflush CR-132
vcd dumpportslimit CR-133
vcd dumpportsoff CR-134
vcd dumpportson CR-135
vcd file CR-136
vcd files CR-138
vcd flush CR-140
vcd limit CR-141
vcd off CR-142
vcd on CR-143
vcom CR-145
vdel CR-151
vdir CR-152
verror CR-153
vgencomp CR-154
view CR-156
virtual count CR-158
virtual define CR-159
virtual delete CR-160
virtual describe CR-161
virtual expand CR-162
virtual function CR-163
virtual hide CR-166
virtual log CR-167
virtual nohide CR-169
virtual nolog CR-170
virtual region CR-172
virtual save CR-173
virtual show CR-174
virtual signal CR-175
virtual type CR-178
vlib CR-180
vlog CR-181
vmake CR-187
vmap CR-188
vsim CR-189
VSIM Tcl commands UM-324
vsimDate CR-203
vsimId CR-203
vsimVersion CR-203
WaveActivateNextPane CR-217
WaveRestoreCursors CR-217
WaveRestoreZoom CR-217
when CR-205
where CR-210
wlf2log CR-211
wlfman CR-213
wlfrecover CR-215
write format CR-216
write list CR-218
write preferences CR-219
write report CR-220
write transcript CR-221
write tssi CR-222
write wave CR-224
comment characters in VSIM commands CR-6
compare simulations UM-117
compatibility, of vendor libraries CR-152
compile history UM-27
compile order
auto generate UM-28
changing UM-28
compiler directives UM-95
IEEE Std 1364-2000 UM-95
XL compatible compiler directives UM-96
compiling
changing order in the GUI UM-28
compile history UM-27
default options, setting UM-240
graphic interface, with the UM-238
grouping files UM-29
options, in projects UM-34
order, changing in projects UM-28
range checking in VHDL CR-148, UM-50
source errors, locating UM-239
Verilog CR-181, UM-69
incremental compilation UM-70
XL 'uselib compiler directive UM-74
XL compatible options UM-73
VHDL CR-145, UM-50
at a specified line number CR-147
selected design units (-just eapbc) CR-146
standard package (-s) CR-148
VITAL packages UM-61
component, default binding rules UM-45
concatenation
directives CR-16
of signals CR-16, CR-175
ConcurrentFileLimit .ini file variable UM-345
conditional breakpoints CR-205, UM-189
configuration simulator state variable UM-353
configurations, simulating CR-189
configure command CR-51
connectivity, exploring UM-156
constants
in case statements CR-147
values of, displaying CR-66, CR-75
context menus
described UM-134
Library tab UM-42
Project tab UM-27

A B C D E F G H I J K L M N O P Q R S T U V W X Y Z

Structure pages UM-201
convert real to time UM-65
convert time to real UM-64
cursors
link to Dataflow window UM-150
locking UM-227
measuring time with UM-227
naming UM-226
trace events with UM-159
Wave window UM-226
customizing
via preference variables UM-352

D

deltas
explained UM-53
Dataflow window UM-149
automatic cell hiding UM-166, UM-167
options UM-166, UM-167
pan UM-158
zoom UM-158
see also windows, Dataflow window
dataflow.bsm file UM-165
dataset alias command CR-55
Dataset Browser UM-121
dataset clear command CR-56
dataset close command CR-57
dataset info command CR-58
dataset list command CR-59
dataset open command CR-60
dataset rename command CR-61, CR-62
Dataset Snapshot UM-123
dataset snapshot command CR-63
datasets UM-117
environment command, specifying with CR-74
managing UM-121
restrict dataset prefix display UM-122
simulator resolution UM-118
DatasetSeparator .ini file variable UM-345
declarations, hiding implicit with explicit CR-149
default binding rules UM-45
default compile options UM-240
default editor, changing UM-337
DefaultForceKind .ini file variable UM-345
DefaultRadix .ini file variable UM-345
DefaultRestartOptions variable UM-346, UM-351
defaults
restoring UM-337
window arrangement UM-134
+define+ CR-181
delay
delta delays UM-53
infinite zero-delay loops, detecting UM-386
interconnect CR-192
modes for Verilog models UM-87
SDF files UM-289
stimulus delay, specifying UM-187
+delay_mode_distributed CR-182
+delay_mode_path CR-182
+delay_mode_unit CR-182
+delay_mode_zero CR-182
'delayed CR-19
DelayFileOpen .ini file variable UM-346
delete command CR-65
deleting library contents UM-41
delta simulator state variable UM-353
deltas
collapsing in the List window UM-176
hiding in the List window CR-52, UM-176
infinite zero-delay loops UM-386
referencing simulator iteration
as a simulator state variable UM-353
dependencies, checking CR-152
dependent design units UM-50
describe command CR-66
descriptions of HDL items UM-197
design hierarchy, viewing in Structure window UM-199
design library
creating UM-40
logical name, assigning UM-43
mapping search rules UM-44
resource type UM-39
VHDL design units UM-50
working type UM-39
design units UM-38
hierarchy of, viewing UM-135
report of units simulated CR-220
Verilog
adding to a library CR-181
directories
mapping libraries CR-188
moving libraries UM-44
disablebp command CR-67
distributed delay mode UM-88
dividers
adding from command line CR-35
Wave window UM-215
DLL files, loading UM-101, UM-102
do command CR-68
DO files (macros) CR-68

A B C D E F G H I J K L M N O P Q R S T U V W X Y Z

error handling UM-333
executing at startup UM-337, UM-347
parameters, passing to UM-331
Tcl source command UM-334
DOPATH environment variable UM-337
drivers
Dataflow Window UM-156
show in Dataflow window UM-218
Wave window UM-218
drivers command CR-69
drivers, multiple on unresolved signal UM-241
dump files, viewing in ModelSim CR-144
dumplog64 command CR-70
dumpports tasks, VCD files UM-304

E

echo command CR-71
edit command CR-72
Editing
in notepad windows UM-147, UM-359
in the Main window UM-147, UM-359
in the Source window UM-147, UM-359
EDITOR environment variable UM-337
editor, default, changing UM-337
elaboration, interrupting CR-189
embedded wave viewer UM-157
enablebp command CR-73
ENDFILE function UM-58
ENDLINE function UM-58
entities
default binding rules UM-45
entities, specifying for simulation CR-201
entity simulator state variable UM-353
enumerated types UM-384
user defined CR-178
environment command CR-74
environment variables UM-337
reading into Verilog code CR-181
referencing from ModelSim command line UM-340
referencing with VHDL FILE variable UM-340
setting in Windows UM-339
specifying library locations in modelsim.ini file UM-341
specifying UNIX editor CR-72
transcript file, specifying location of UM-348
using in pathnames CR-12
using with location mapping UM-387
variable substitution using Tcl UM-323
viewing current names and values with printenv CR-103
environment, displaying or changing pathname CR-74
errors
bad magic number UM-119
during compilation, locating UM-239
getting details about messages CR-153
onerror command CR-100
event order
changing in Verilog CR-181
in Verilog simulation UM-79
event queues UM-79
events, tracing UM-159
examine command CR-75
examine tooltip
toggling on/off UM-223
exit command CR-78
expand net UM-156
Explicit .ini file variable UM-342
Expression Builder UM-262
configuring a List trigger with UM-382
extended identifiers CR-14
syntax in commands CR-12

F

-f CR-182
file I/O
TextIO package UM-55
VCD files UM-303
file-line breakpoints UM-197
files, grouping for compile UM-29
filtering signals in Signals window UM-185
find command CR-79
finding
cursors in the Wave window UM-227
marker in the List window UM-178
names and values UM-133
folders, in projects UM-32
force command CR-82
defaults UM-351
format file
List window CR-216
Wave window CR-216, UM-208
FPGA libraries, importing UM-48

G

GenerateFormat .ini file variable UM-346
generics
assigning or overriding values with -g and -G CR-

A B C D E F G H I J K L M N O P Q R S T U V W X Y Z

190
examining generic values CR-75
limitation on assigning composite types CR-191
get_resolution() VHDL function UM-62
glitches
disabling generation
from command line CR-196
from GUI UM-248
graphic interface UM-129
grouping files for compile UM-29
GUI preferences, saving UM-352
GUI_expression_format CR-15
GUI expression builder UM-262
syntax CR-18

H

'hasX CR-19
Hazard .ini file variable (VLOG) UM-343
hazards
-hazards argument to vlog CR-182
-hazards argument to vsim CR-197
limitations on detection UM-82
HDL item UM-16
help command CR-85
hierarchy
forcing signals in UM-63
referencing signals in UM-63
releasing signals in UM-63
viewing signal names without UM-222
history
of commands
shortcuts for reuse CR-7, UM-358
of compiles UM-27
history command CR-86
HOME environment variable UM-337

I

I/O
TextIO package UM-55
VCD files UM-303
ieee .ini file variable UM-341
IEEE libraries UM-46
IEEE Std 1076 UM-14
IEEE Std 1364 UM-14, UM-68
IgnoreError .ini file variable UM-346
IgnoreFailure .ini file variable UM-346
IgnoreNote .ini file variable UM-346
IgnoreVitalErrors .ini file variable UM-342
IgnoreWarning .ini file variable UM-346
implicit operator, hiding with vcom -explicit CR-149
importing FPGA libraries UM-48
+incdir+ CR-182
incremental compilation
automatic UM-71
manual UM-71
with Verilog UM-70
index checking UM-50
init_signal_spy UM-63
init_usertfs function UM-98
initial dialog box, turning on/off UM-336
interconnect delays CR-192, UM-300
annotating per Verilog 2001 CR-200
internal signals, adding to a VCD file CR-127
item_list_file, WLF files CR-213
iteration_limit, infinite zero-delay loops UM-386
IterationLimit .ini file variable UM-346

K

keyboard shortcuts
List window UM-180, UM-357
Main window UM-147, UM-359
Source window UM-359
Wave window UM-231, UM-356

L

language templates UM-264
libraries
archives CR-180
dependencies, checking CR-152
design libraries, creating CR-180, UM-40
design library types UM-39
design units UM-38
group use, setting up UM-379
IEEE UM-46
importing FPGA libraries UM-48
including precompiled modules UM-250
listing contents CR-152
mapping
from the command line UM-43
from the GUI UM-43
hierarchically UM-349
search rules UM-44
modelsim_lib UM-62
moving UM-44
multiple libraries with common modules UM-72
naming UM-43

A B C D E F G H I J K L M N O P Q R S T U V W X Y Z

predefined UM-46
refreshing library images CR-148, CR-184, UM-47
resource libraries UM-39
std library UM-46
Synopsys UM-47
vendor supplied, compatibility of CR-152
Verilog CR-197, UM-72
VHDL library clause UM-45
working libraries UM-39
working with contents of UM-41
library simulator state variable UM-353
License variable in .ini file UM-347
licensing
License variable in .ini file UM-347
lint-style checks CR-183
List window UM-168
adding items to CR-32
setting triggers UM-382
see also windows, List window
LM_LICENSE_FILE environment variable UM-337
location maps, referencing source files UM-387
log command CR-87
log file
log command CR-87
nolog command CR-93
overview UM-117
QuickSim II format CR-211
redirecting with -l CR-192
virtual log command CR-167
virtual nolog command CR-170
see also WLF files
lshift command CR-89
lsublist command CR-90

M

MacroNestingLevel simulator state variable UM-353
macros (DO files) UM-331
breakpoints, executing at CR-47
creating from a saved transcript UM-139
depth of nesting, simulator state variable UM-353
error handling UM-333
executing CR-68
forcing signals, nets, or registers CR-82
parameters
as a simulator state variable (n) UM-353
passing CR-68, UM-331
total number passed UM-353
relative directories CR-68
shifting parameter values CR-118
startup macros UM-350
Main window UM-137
see also windows, Main window
mapping
libraries
from the command line UM-43
hierarchically UM-349
symbols
Dataflow window UM-165
mapping libraries, library mapping UM-43
math_complex package UM-47
math_real package UM-47
+maxdelays CR-183
mc_scan_plusargs, PLI routine CR-199
memory
modeling in VHDL UM-390
menus
Dataflow window UM-150
List window UM-170
Main window UM-140
Process window UM-182
Signals window UM-184
Source window UM-192
Structure window UM-200
tearing off or pinning menus UM-134
Variables window UM-204
Wave window UM-209
messages
bad magic number UM-119
echoing CR-71
getting more information CR-153
loading, disbling with -quiet CR-148, CR-183
redirecting UM-348
suppressing warnings from arithmetic packages UM-350
turning off assertion messages UM-350
MGC_LOCATION_MAP variable UM-337
+mindelays CR-183
mnemonics, assigning to signal values CR-178
MODEL_TECH environment variable UM-337
MODEL_TECH_TCL environment variable UM-337
modeling memory in VHDL UM-390
ModelSim
commands CR-23–CR-212
modelsim command CR-91
MODELSIM environment variable UM-338
modelsim.ini
default to VHDL93 UM-351
delay file opening with UM-351
environment variables in UM-349
force command default, setting UM-351

A B C D E F G H I J K L M N O P Q R S T U V W X Y Z

hierarchical library mapping UM-349
opening VHDL files UM-351
restart command defaults, setting UM-351
startup file, specifying with UM-350
transcript file created from UM-349
turning off arithmetic package warnings UM-350
turning off assertion messages UM-350
modelsim.tcl file UM-352
modelsim_lib UM-62
path to UM-341
MODELSIM_TCL environment variable UM-338
Modified field, Project tab UM-26
modules
handling multiple, common names UM-72
mouse shortcuts
Main window UM-147, UM-359
Source window UM-359
Wave window UM-231, UM-356
.mpf file UM-18
loading from the command line UM-35
mti_cosim_trace environment variable UM-338
MTI_TF_LIMIT environment variable UM-338
multiple drivers on unresolved signal UM-241
multiple simulations UM-117
multi-source interconnect delays CR-192

N

n simulator state variable UM-353
name case sensitivity, VHDL vs. Verilog CR-12
Name field
Project tab UM-26
negative pulses
driving an error state CR-200
negative timing
$setuphold/$recovery UM-92
algorithm for calculating delays UM-83
check limits UM-83
extending check limits CR-197
nets
adding to the Wave and List windows UM-187
Dataflow window, displaying in UM-149
drivers of, displaying CR-69
stimulus CR-82
values of
displaying in Signals window UM-183
examining CR-75
forcing UM-186
saving as binary log file UM-187
waveforms, viewing UM-206
next and previous edges, finding UM-232, UM-357
Nlview widget Symlib format UM-165
no space in time literal UM-241
NoCaseStaticError .ini file variable UM-342
NoDebug .ini file variable (VCOM) UM-342
NoDebug .ini file variable (VLOG) UM-343
noforce command CR-92
NoIndexCheck .ini file variable UM-342
+nolibcell CR-183
nolog command CR-93
NOMMAP environment variable UM-338
non-blocking assignments UM-81
NoOthersStaticError .ini file variable UM-342
NoRangeCheck .ini file variable UM-342
notepad command CR-95
Notepad windows, text editing UM-147, UM-359
-notrigger argument UM-381
noview command CR-96
NoVital .ini file variable UM-342
NoVitalCheck .ini file variable UM-342
Now simulator state variable UM-353
now simulator state variable UM-353
+nowarn<CODE> CR-183
nowhen command CR-97
numeric_bit package UM-47
numeric_std package UM-47
disabling warning messages UM-350
NumericStdNoWarnings .ini file variable UM-347

O

onbreak command CR-98
onElabError command CR-99
onerror command CR-100
optimize for std_logic_1164 UM-242
Optimize_1164 .ini file variable UM-342
OptionFile entry in project files UM-244
order of events
changing in Verilog CR-181
ordering files for compile UM-28
organizing projects with folders UM-32
others .ini file variable UM-342

P

packages
standard UM-46
textio UM-46
util UM-62
VITAL 1995 UM-60

A B C D E F G H I J K L M N O P Q R S T U V W X Y Z

VITAL 2000 UM-60
page setup
Dataflow window UM-164
Wave window UM-236
pan, Dataflow window UM-158
parameters
making optional UM-332
using with macros CR-68, UM-331
path delay mode UM-88
pathnames
in VSIM commands CR-10
spaces in CR-9
PathSeparator .ini file variable UM-347
pause command CR-101
PedanticErrors .ini file variable UM-342
PLI
specifying which apps to load UM-98
Veriuser entry UM-98
PLI/VPI UM-97
tracing UM-113
PLIOBJS environment variable UM-98, UM-338
popup
toggling waveform popup on/off UM-223
port driver data, capturing UM-312
Postscript
saving a waveform in UM-233
saving the Dataflow display in UM-162
precedence of variables UM-353
precision, simulator resolution UM-77
pref.tcl file UM-352
preference variables
.ini files, located in UM-341
editing UM-352
saving UM-352
Tcl files, located in UM-352
preferences, saving UM-352
primitives, symbols in Dataflow window UM-165
printenv command CR-102, CR-103
Process window UM-181
see also windows, Process window
processes
values and pathnames in Variables window UM-203
without wait statements UM-241
Programming Language Interface UM-97
project context menus UM-27
project tab
information in UM-26
sorting UM-26
projects UM-17
accessing from the command line UM-35
adding files to UM-21
benefits UM-18
compile order UM-28
changing UM-28
compiler options in UM-34
compiling files UM-24
context menu UM-27
creating UM-20
creating simulation configurations UM-30
differences with earlier versions UM-19
folders in UM-32
grouping files in UM-29
loading a design UM-25
MODELSIM environment variable UM-338
override mapping for work directory with vcom CR-149
override mapping for work directory with vlog CR-185
overview UM-18
propagation, preventing X propagation CR-192
pulse error state CR-200
pwd command CR-105

Q

QuickSim II logfile format CR-211
Quiet .ini file variable
VCOM UM-342
Quiet .ini file variable (VLOG) UM-343
quietly command CR-106
quit command CR-107

R

race condition, problems with event order UM-79
radix
changing in Signals, Variables, Dataflow, List, and Wave windows CR-108
character strings, displaying CR-178
default, DefaultRadix variable UM-345
of signals being examined CR-76
of signals in Wave window CR-37
specifying in List window UM-173
radix command CR-108
range checking UM-50
disabling CR-147
enabling CR-148
readers and drivers UM-156
real type, converting to time UM-65
reconstruct RTL-level design busses UM-126

A B C D E F G H I J K L M N O P Q R S T U V W X Y Z

record field selection, syntax CR-10
records, values of, changing UM-203
$recovery UM-92
redirecting messages, TranscriptFile UM-348
refreshing library images CR-148, CR-184, UM-47
registers
  adding to the Wave and List windows UM-187
  values of
    displaying in Signals window UM-183
    saving as binary log file UM-187
  waveforms, viewing UM-206
report
  simulator control UM-336
  simulator state UM-336
report command CR-109
reporting
  compile history UM-27
  variable settings CR-13
RequireConfigForAllDefaultBinding variable UM-342
resolution
  returning as a real UM-62
  specifying with -t argument CR-193
  verilog simulation UM-77
  VHDL simulation UM-52
Resolution .ini file variable UM-347
resolution simulator state variable UM-353
resource libraries UM-45
restart command CR-111
  defaults UM-351
  in GUI UM-142
  toolbar button UM-145, UM-195, UM-214
restoring defaults UM-337
results, saving simulations UM-117
resume command CR-113
RTL-level design busses
  reconstructing UM-126
run command CR-114
RunLength .ini file variable UM-347

S

saving
  simulation options in a project UM-30
  waveforms UM-117
scope, setting region environment CR-74
SDF
  disabling timing checks UM-300
  errors and warnings UM-291
  instance specification UM-290
  interconnect delays UM-300
  mixed VHDL and Verilog designs UM-300
  specification with the GUI UM-291
  troubleshooting UM-301
  Verilog
    $sdf_annotate system task UM-294
    optional conditions UM-299
    optional edge specifications UM-298
    rounded timing values UM-299
    SDF to Verilog construct matching UM-295
  VHDL
    resolving errors UM-293
    SDF to VHDL generic matching UM-292
$sdf_done UM-94
search libraries CR-197, UM-250
searching
  in the source window UM-197
  in the Structure window UM-202
  List window
    signal values, transitions, and names UM-177
  values and names UM-133
  Verilog libraries UM-72
  Wave window
    signal values, edges and names UM-225
searchlog command CR-116
$setuphold UM-92
shared objects
  loading FLI applications
    see ModelSim FLI Reference manual
  loading PLI/VPI C applications UM-101
  loading PLI/VPI C++ applications UM-102
shift command CR-118
Shortcuts
  text editing UM-147, UM-359
shortcuts
  command history CR-7, UM-358
  command line caveat CR-7, UM-358
  List window UM-180, UM-357
  Main window UM-359
  Main windows UM-147
  Source window UM-359
  Wave window UM-231, UM-356
show command CR-119
show drivers
  Dataflow window UM-156
  Wave window UM-218
show source lines with errors UM-241
Show_Lint .ini file variable (VLOG) UM-343
Show_source .ini file variable
  VCOM UM-342
Show_source .ini file variable (VLOG) UM-343
Show_VitalChecksWarning .ini file variable UM-342

A B C D E F G H I J K L M N O P Q R S T U V W X Y Z

Show_Warning1 .ini file variable UM-343
Show_Warning2 .ini file variable UM-343
Show_Warning3 .ini file variable UM-343
Show_Warning4 .ini file variable UM-343
Show_Warning5 .ini file variable UM-343
Signal Spy UM-63
signal_force UM-63
signal_release UM-63
signals
  adding to a WLF file UM-187
  adding to the Wave and List windows UM-187
  alternative names in the List window (-label) CR-33
  alternative names in the Wave window (-label) CR-36
  applying stimulus to UM-186
  attributes of, using in expressions CR-19
  breakpoints CR-205, UM-189
  combining into a user-defined bus CR-36, UM-174, UM-217
  Dataflow window, displaying in UM-149
  drivers of, displaying CR-69
  environment of, displaying CR-74
  filtering in the Signals window UM-185
  finding CR-79
  force time, specifying CR-83
  hierarchy
    referencing in UM-63
    releasing in UM-63
  log file, creating CR-87
  names of, viewing without hierarchy UM-222
  pathnames in VSIM commands CR-10
  radix
    specifying for examine CR-76
    specifying in List window CR-33
    specifying in Wave window CR-37
  sampling at a clock change UM-381
  states of, displaying as mnemonics CR-178
  stimulus CR-82
  transitions, searching for UM-228
  types, selecting which to view UM-185
  unresolved, multiple drivers on UM-241
  values of
    converting to strings UM-384
    displaying in Signals window UM-183
    examining CR-75
    forcing anywhere in the hierarchy UM-63
    replacing with text CR-178
    saving as binary log file UM-187
  waveforms, viewing UM-206
Signals window UM-183
  see also windows, Signals window
simulating
  command-line mode UM-378
  comparing simulations UM-117
  default run length UM-255
  delays, specifying time units for CR-14
  design unit, specifying CR-189
  graphic interface to UM-245
  iteration limit UM-255
  saving dataflow display as a Postscript file UM-162
  saving options in a project UM-30
  saving simulations CR-87, CR-194, UM-117, UM-379
  saving waveform as a Postscript file UM-233
  stepping through a simulation CR-122
  stimulus, applying to signals and nets UM-186
  stopping simulation in batch mode CR-208
  time resolution UM-246
  Verilog UM-76
    delay modes UM-87
    hazard detection UM-82
    resolution limit UM-77
    XL compatible simulator options UM-86
  VHDL UM-52
  viewing results in List window UM-168
  VITAL packages UM-61
Simulation Configuration
  creating UM-30
simulations
  event order in UM-79
  saving results CR-62, CR-63, UM-117
  saving results at intervals UM-123
simulator resolution
  returning as a real UM-62
  Verilog UM-77
  VHDL UM-52
  vsim -t argument CR-193
  when comparing datasets UM-118
simulator state variables UM-353
simulator version CR-193, CR-203
simultaneous events in Verilog
  changing order CR-181
sizetf callback function UM-107
so, shared object file
  loading PLI/VPI C applications UM-101
  loading PLI/VPI C++ applications UM-102
software version UM-144
sorting
  HDL items in GUI windows UM-133
source directory, setting from source window UM-192
source errors, locating during compilation UM-239
source files, referencing with location maps UM-387

A B C D E F G H I J K L M N O P Q R S T U V W X Y Z

source libraries
    arguments supporting UM-73
source lines with errors
    showing UM-241
spaces in pathnames CR-9
specify path delays CR-200
standards supported UM-14
startup
    alternate to startup.do (vsim -do) CR-190
    macro in the modelsim.ini file UM-347
    macros UM-350
    using a startup file UM-350
Startup .ini file variable UM-347
state variables UM-353
status bar
    Main window UM-147
status command CR-121
Status field
    Project tab UM-26
std .ini file variable UM-341
std_arith package
    disabling warning messages UM-350
std_developerskit .ini file variable UM-341
std_logic_arith package UM-47
std_logic_signed package UM-47
std_logic_textio UM-47
std_logic_unsigned package UM-47
StdArithNoWarnings .ini file variable UM-347
STDOUT environment variable UM-338
step command CR-122
stimulus
    applying to signals and nets UM-186
stop command CR-123
Structure window UM-199
    see also windows, Structure window
symbol mapping
    Dataflow window UM-165
symbolic constants, displaying CR-178
symbolic names, assigning to signal values CR-178
synopsys .ini file variable UM-341
Synopsys libraries UM-47
synthesis
    rule compliance checking CR-145, UM-242, UM-342
system calls
    VCD UM-304
    Verilog UM-89
system commands UM-323
system tasks
    ModelSim Verilog UM-94
    VCD UM-304
    Verilog UM-89
    Verilog-XL compatible UM-92

T

tab stops, in the Source window UM-198
tb command CR-124
Tcl UM-315–UM-326
    command separator UM-322
    command substitution UM-321
    command syntax UM-318
    evaluation order UM-322
    Man Pages in Help menu UM-144
    preference variables UM-352
    relational expression evaluation UM-322
    time commands UM-325
    variable
        in when commands CR-206
        substitution UM-323
    VSIM Tcl commands UM-324
temp files, VSOUT UM-340
text and command syntax UM-16
Text editing UM-147, UM-359
TextIO package
    alternative I/O files UM-59
    containing hexadecimal numbers UM-58
    dangling pointers UM-58
    ENDFILE function UM-58
    ENDLINE function UM-58
    file declaration UM-55
    implementation issues UM-57
    providing stimulus UM-59
    standard input UM-56
    standard output UM-56
    WRITE procedure UM-57
    WRITE_STRING procedure UM-57
TF routines UM-111
TFMPC
    disabling warning CR-199
time
    absolute, using @ CR-14
    simulation time units CR-14
    time resolution as a simulator state variable UM-353
time literal, missing space UM-241
time resolution
    in Verilog UM-77
    in VHDL UM-52
    setting
        with the GUI UM-246
        with vsim command CR-193

A B C D E F G H I J K L M N O P Q R S T U V W X Y Z

time type, converting to real UM-64
time, time units, simulation time CR-14
time-based breakpoints UM-189
timescale directive warning, disabling CR-199
timing
  $setuphold/$recovery UM-92
  annotation UM-289
  disabling checks CR-183, UM-300
  disabling checks for entire design CR-192
  negative check limits
    described UM-83
    extending CR-197
title, Main window, changing CR-193
to_real VHDL function UM-64
to_time VHDL function UM-65
toggling waveform popup on/off UM-223
toolbar
  Dataflow window UM-153
  Main window UM-145
  Wave window UM-212
tooltip, toggling waveform popup UM-223
tracing
  events UM-159
  source of unknown UM-160
transcript
  file name, specifed in modelsim.ini UM-349
  saving UM-139
  TranscriptFile variable in .ini file UM-348
  using as a DO file UM-139
transcript command CR-125
transcript file
  redirecting with -l CR-192
tree windows
  VHDL and Verilog items in UM-135
  viewing the design hierarchy UM-136
TreeUpdate command CR-217
triggers, in the List window UM-382
triggers, in the List window, setting UM-176
TSCALE, disabling warning CR-199
TSSI CR-222
  in VCD files UM-312
tssi2mti command CR-126
type
  converting real to time UM-65
  converting time to real UM-64
Type field, Project tab UM-26

U

-u CR-184
unbound component UM-241
UnbufferedOutput .ini file variable UM-348
unit delay mode UM-88
unknowns, tracing UM-160
unresolved signals, multiple drivers on UM-241
use 1076-1993 language standard UM-240
use clause, specifying a library UM-46
use explicit declarations only UM-241
user-defined bus CR-36, UM-125, UM-174, UM-217
UserTimeUnit .ini file variable UM-348
util package UM-62

V

-v CR-184
v2k_int_delays CR-200
values
  describe HDL items CR-66
  examine HDL item values CR-75
  of HDL items UM-197
  replacing signal values with strings CR-178
variable settings report CR-13
variables
  adding to the Wave and List windows UM-187
  describing CR-66
  environment variables UM-337
  LM_LICENSE_FILE UM-337
  personal preferences UM-336
  precedence between .ini and .tcl UM-353
  setting environment variables UM-337
  simulator state variables
    current settings report UM-336
    iteration number UM-353
    name of entity or module as a variable UM-353
    resolution UM-353
    simulation time UM-353
  value of
    changing from command line CR-50
    changing with the GUI UM-203
    examining CR-75
  values of
    displaying in Signals window UM-183
    saving as binary log file UM-187
Variables window UM-203
  see also windows, Variables window
vcd add command CR-127
vcd checkpoint command CR-128
vcd comment command CR-129
vcd dumpports command CR-130
vcd dumportsall command CR-131

A B C D E F G H I J K L M N O P Q R S T U V W X Y Z

vcd dumpportsflush command CR-132
vcd dumpportslimit command CR-133
vcd dumpportsoff command CR-134
vcd dumpportson command CR-135
vcd file command CR-136
VCD files UM-303
  adding items to the file CR-127
  capturing port driver data CR-130, UM-312
  case sensitivity UM-306
  converting to WLF files CR-144
  creating CR-127, UM-306
  dumping variable values CR-128
  dumpports tasks UM-304
  flushing the buffer contents CR-140
  from VHDL source to VCD output UM-309
  inserting comments CR-129
  internal signals, adding CR-127
  specifying maximum file size CR-141
  specifying name of CR-138
  specifying the file name CR-136
  state mapping CR-136, CR-138
  supported TSSI states UM-312
  turn off VCD dumping CR-142
  turn on VCD dumping CR-143
  VCD system tasks UM-304
  viewing files from another tool CR-144
vcd files command CR-138
vcd flush command CR-140
vcd limit command CR-141
vcd off command CR-142
vcd on command CR-143
vcd2wlf command CR-144
vcom command CR-145
vdel command CR-151
vdir command CR-152
vector elements, initializing CR-50
vendor libraries, compatibility of CR-152
Vera, see Vera documentation
Verilog
  ACC routines UM-110
  capturing port driver data with -dumpports CR-136, UM-312
  cell libraries UM-87
  compiler directives UM-95
  compiling and linking PLI C applications UM-101
  compiling and linking PLI C++ applications UM-102
  compiling design units UM-69
  compiling with XL 'uselib compiler directive UM-74
  creating a design library UM-69
  event order in simulation UM-79
  language templates UM-264
  library usage UM-72
  SDF annotation UM-294
  sdf_annotate system task UM-294
  simulating UM-76
    delay modes UM-87
    XL compatible options UM-86
  simulation hazard detection UM-82
  simulation resolution limit UM-77
  source code viewing UM-191
  standards UM-14
  system tasks UM-89
  TF routines UM-111
  XL compatible compiler options UM-73
  XL compatible routines UM-113
  XL compatible system tasks UM-92
verilog .ini file variable UM-341
Verilog 2001
  current implementation UM-14, UM-68
  disabling support CR-184
Verilog PLI/VPI UM-97–UM-115
  64-bit support in the PLI UM-113
  compiling and linking PLI/VPI C applications UM-101
  compiling and linking PLI/VPI C++ applications UM-102
  debugging PLI/VPI code UM-113
  PLI callback reason argument UM-106
  PLI support for VHDL objects UM-109
  registering PLI applications UM-97
  registering VPI applications UM-99
  specifying the PLI/VPI file to load UM-103
Verilog-XL
  compatibility with UM-67
Veriuser .ini file variable UM-98, UM-348
Veriuser, specifying PLI applications UM-98
veriuser.c file UM-108
verror command CR-153
version
  obtaining via Help menu UM-144
  obtaining with vsim command CR-193
  obtaining with vsim<info> commands CR-203
vgencomp command CR-154
VHDL
  delay file opening UM-351
  dependency checking UM-50
  field naming syntax CR-10
  file opening delay UM-351
  language templates UM-264
  library clause UM-45

A B C D E F G H I J K L M N O P Q R S T U V W X Y Z

object support in PLI UM-109
simulating UM-52
source code viewing UM-191
standards UM-14
timing check disabling UM-52
VITAL package UM-47
VHDL utilities UM-62, UM-63
get_resolution() UM-62
to_real() UM-64
to_time() UM-65
VHDL93 .ini file variable UM-343
view command CR-156
viewing
design hierarchy UM-135
library contents UM-41
waveforms CR-194, UM-117
virtual count commands CR-158
virtual define command CR-159
virtual delete command CR-160
virtual describe command CR-161
virtual expand commands CR-162
virtual function command CR-163
virtual hide command CR-166, UM-126
virtual log command CR-167
virtual nohide command CR-169
virtual nolog command CR-170
virtual objects UM-125
virtual functions UM-126
virtual regions UM-127
virtual signals UM-125
virtual types UM-127
virtual region command CR-172, UM-127
virtual regions
reconstruct the RTL hierarchy in gate-level design UM-127
virtual save command CR-173, UM-126
virtual show command CR-174
virtual signal command CR-175, UM-125
virtual signals
reconstruct RTL-level design busses UM-126
reconstruct the original RTL hierarchy UM-126
virtual hide command UM-126
virtual type command CR-178
VITAL
compiling and simulating with accelerated VITAL packages UM-61
disabling optimizations for debugging UM-61
specification and source code UM-60
VITAL packages UM-60
vital95 .ini file variable UM-341
vlib command CR-180
vlog command CR-181
vlog.opt file UM-244
vlog95compat .ini file variable UM-343
vmake command CR-187
vmap command CR-188
VPI, registering applications UM-99
VPI/PLI UM-97
compiling and linking C applications UM-101
compiling and linking C++ applications UM-102
vsim build date and version CR-203
vsim command CR-189
VSOUT temp file UM-340

W

WARNING[8], -lint argument to vlog CR-183
warnings
disabling at time 0 UM-350
locating file and line number UM-380
suppressing VCOM warning messages CR-148
suppressing VLOG warning messages CR-183
suppressing VSIM warning messages CR-199
turning off warnings from arithmetic packages UM-350
wave format file UM-208
wave log format (WLF) file CR-194, UM-117
of binary signal values CR-87
see also WLF files
wave viewer, Dataflow window UM-157
Wave window UM-206
in the Dataflow window UM-157
toggling waveform popup on/off UM-223
see also windows, Wave window
wave, adding CR-35
WaveActivateNextPane command CR-217
waveform logfile
log command CR-87
overview UM-117
see also WLF files
waveform popup UM-223
waveforms UM-117
saving and viewing CR-87, UM-118
saving and viewing in batch mode UM-379
viewing UM-206
WaveRestoreCursors command CR-217
WaveRestoreZoom command CR-217
WaveSignalNameWidth .ini file variable UM-348
welcome dialog, turning on/off UM-336
when command CR-205
when statement

A B C D E F G H I J K L M N O P Q R S T U V W X Y Z

setting signal breakpoints UM-189
time-based breakpoints CR-209
where command CR-210
wildcard characters
for pattern matching in simulator commands CR-13
Windows
Main window
text editing UM-147, UM-359
Source window
text editing UM-147, UM-359
windows
Dataflow window UM-149
toolbar UM-153
zooming UM-158
finding HDL item names in UM-133
List window UM-168
adding HDL items UM-169
adding signals with a WLF file UM-187
display properties of UM-175
formatting HDL items UM-172
output file CR-218
saving data to a file UM-179
saving the format of CR-216
setting triggers UM-176, UM-382
time markers UM-133
Main window UM-137
status bar UM-147
time and delta display UM-147
toolbar UM-145
opening
from command line CR-156
with the GUI UM-141
Process window UM-181
displaying active processes UM-181
specifying next process to be executed UM-181
viewing processing in the region UM-181
saving position and size UM-134
searching for HDL item values in UM-133
Signals window UM-183
VHDL and Verilog items viewed in UM-183
Source window
setting tab stops UM-198
Structure window UM-199
selecting items to view in Signals window UM-183
VHDL and Verilog items viewed in UM-199
viewing design hierarchy UM-199
Variables window UM-203
VHDL and Verilog items viewed in UM-203
Wave window UM-206
adding HDL items to UM-208
adding signals with a WLF file UM-187
cursor measurements UM-227
display properties UM-222
display range (zoom), changing UM-228
format file, saving UM-208
path elements, changing CR-53, UM-348
time cursors UM-226
zooming UM-228
WLF files
adding items to UM-187
creating from VCD CR-144
filtering, combining CR-213
limiting size CR-194
log command CR-87
overview UM-118
repairing CR-215
saving CR-62, CR-63, UM-119
saving at intervals UM-123
specifying name CR-194
using in batch mode UM-379
wlf2log command CR-211
wlfman command CR-213
wlfrecover command CR-215
work library UM-39
workspace UM-138
write format command CR-216
write list command CR-218
write preferences command CR-219
write report command CR-220
write transcript command CR-221
write tssi command CR-222
write wave command CR-224

X

X
tracing unknowns UM-160
X propagation
disabling for entire design CR-192

Y

-y CR-185

Z

zero delay elements UM-53
zero delay mode UM-88
zero-delay loop, infinite UM-386

A B C D E F G H I J K L M N O P Q R S T U V W X Y Z

zero-delay oscillation UM-386
zero-delay race condition UM-79
zoom
    Dataflow window UM-158
    from Wave toolbar buttons UM-228
    saving range with bookmarks UM-229
    with the mouse UM-229